João Bernardo Coimbra Marques

Bachelor of Science in Computer Science and Engineering

Privacy-preserving key-value store

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Nuno Manuel Ribeiro Preguiça, Associate Professor,
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa

Co-adviser: Bernardo Luís da Silva Ferreira, Assistant Professor,
Faculdade de Ciências
Universidade de Lisboa

Examination Committee:

Chair: João Baptista da Silva Araújo Júnior,
Associate Professor, FCT/UNL

Rapporteur: Miguel Matos, Assistant Professor, IST/UL

Member: Nuno Manuel Ribeiro Preguiça,
Associate Professor, FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Privacy-preserving key-value store

Acknowledgements

I would like to start by thanking my advisor, Professor Nuno Preguiça, and co-advisor, Professor Bernardo Ferreira, for their guidance, support and dedication. Without them this work would not have been possible.

I also want to thank Bernardo Portela, Annette Bieniusa, Gonçalo Tomás and Pedro Lopes for answering my questions and helping me with work related to this thesis.

Finally, thanks to my family and friends for their continued support.

I would like to acknowledge Fundação para a Ciência e Tecnologia (FCT/MCTES) for their funding through the SAMOA project (PTDC/CCI-INF/32662/2017), and Departamento de Informática (DI-FCT-UNL) and NOVA LINCS for allowing me to use DI-Cluster.

Abstract

Cloud computing is arguably the foremost delivery platform for data storage and data processing. It turned computing into a utility based service that provides consumers and enterprises with on-demand access to computing resources. Although advantageous, there is an inherent lack of control over the hardware in the cloud computing model, this may constitute an increased privacy and security risk.

Multiple encrypted database systems have emerged in recent years, they provide the functionality of regular databases but without compromising data confidentiality. These systems leverage novel encryption schemes such as homomorphic and searchable encryption. However, many of these proposals focus on extending existing centralized systems that are very difficult to scale, and offer poor performance in geo-replicated scenarios.

We propose a scalable, highly available, and geo-replicated privacy-preserving key-value store. A system that provides its users with secure data types meant to be replicated, along with a rich query interface with configurable privacy that enables one to issue secure and somewhat complex queries. We accompany our proposal with an implementation of a privacy-preserving client library for AntidoteDB, a geo-replicated key-value store. We also extend the AntidoteDB's query language interface by adding support for secure SQL-like queries with configurable privacy. Experimental evaluations show that our proposals offer a feasible solution to practical applications that wish to improve their privacy and confidentiality.

Keywords: Cloud Computing, Key-Value Stores, CRDT, Privacy, Homomorphic Encryption, Searchable Encryption.

Resumo

A computação na nuvem é sem dúvida a principal plataforma de serviços de armazenamento e processamento de dados. Transformou a indústria da computação num serviço básico que oferece aos seus utilizadores um acesso flexível a recursos computacionais. Apesar das vantagens, existe uma inerente perda de acesso e controlo sobre o equipamento físico que pode introduzir riscos acrescidos de segurança e privacidade.

Nos últimos anos têm surgido múltiplas propostas de sistemas de bases de dados seguros, capazes de oferecer todas as funcionalidades esperadas, sem comprometer a segurança e privacidade dos dados armazenados. Estes sistemas utilizam esquemas criptográficos inovadores baseados em cifras homomórficas e cifras pesquisáveis. Contudo, a maioria destas propostas são baseadas em sistemas de bases de dados principalmente centralizados. Sistemas difíceis de escalar com um desempenho pobre em cenários geo-replicados.

A tese propõe um sistema de armazenamento chave-valor escalável, altamente disponível e geo-replicado. Um sistema que oferece aos seus utilizadores tipos de dados replicados seguros com uma interface de consulta rica, e que permite à aplicação escolher diferentes níveis de segurança e privacidade. A nossa proposta é acompanhada por uma implementação de uma biblioteca cliente segura para o AntidoteDB, um sistema de armazenamento chave-valor geo-replicado. O nosso trabalho inclui ainda a modificação da interface de consulta rica do AntidoteDB, adicionando a possibilidade de executar consultas com diferentes níveis de segurança e privacidade. Finalmente, apresentamos um estudo experimental que mostra que as nossas propostas oferecem uma solução prática a aplicações que pretendem melhorar a sua privacidade e confidencialidade.

Palavras-chave: Computação na Nuvem, Sistemas de Armazenamento Chave-Valor, CRDT, Privacidade, Cifras Homomórficas, Cifras Pesquisáveis.

Contents

Table of contents xi

List of Figures xiii

List of Tables xv

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Solution . . . 2
- 1.3 Contributions . . . 3
- 1.4 Outline . . . 3

2 Related Work 5
- 2.1 Computations on ciphertexts . . . 5
  - 2.1.1 Homomorphic encryption . . . 5
  - 2.1.2 Property-preserving encryption . . . 6
  - 2.1.3 Searchable encryption . . . 7
  - 2.1.4 Discussion . . . 8
- 2.2 Secure data stores . . . 9
  - 2.2.1 CryptDB . . . 9
  - 2.2.2 Cipherbase . . . 12
  - 2.2.3 Arx . . . 13
  - 2.2.4 Privacy-preserving NoSQL databases . . . 15
  - 2.2.5 Discussion . . . 16
- 2.3 Conflict-free replicated data types . . . 16
  - 2.3.1 State-based synchronization . . . 17
  - 2.3.2 Operation-based synchronization . . . 18
  - 2.3.3 Delta-based synchronization . . . 19
  - 2.3.4 Concurrency semantics . . . 19
- 2.4 Summary . . . 20

3 Secure Conflict-free Replicated Data Types 21
- 3.1 Black-box constructions . . . 21

3.2 Homomorphic constructions . . . 23
3.3 Summary . . . 25

4 Integration of SCRDTs into AntidoteDB 27
4.1 AntidoteDB . . . 27
4.2 Design and implementation . . . 29
4.2.1 Client libraries . . . 30
4.2.2 Modifications to AntidoteDB . . . 32
4.3 Summary . . . 33

5 SCRDTs experimental evaluation 35
5.1 Synthetic benchmarks . . . 36
5.1.1 Register . . . 36
5.1.2 Set . . . 37
5.1.3 Counter . . . 38
5.1.4 Bounded counter . . . 40
5.1.5 Discussion . . . 41
5.2 Realistic benchmark . . . 41
5.2.1 Results . . . 42
5.3 Summary . . . 44

6 Antidote Query Language 45
6.1 AQL's architecture . . . 45
6.2 Design and implementation of a secure AQL . . . 46
6.2.1 Query rewriter . . . 48
6.2.2 Indexing system . . . 49
6.2.3 Modifications to the AQL module . . . 49
6.3 Summary . . . 50

7 AQL experimental evaluation 51
7.1 Setup . . . 51
7.2 Benchmark . . . 51
7.3 Results . . . 53
7.3.1 FMKe . . . 55
7.4 Summary . . . 55

8 Conclusion 57
8.1 Future work . . . 58
8.2 Publications . . . 58

Bibliography 59

List of Figures

2.1 CryptDB's architecture . . . 10
2.2 Cipherbase's architecture . . . 12
2.3 Arx's architecture . . . 14

4.1 AntidoteDB's general architecture . . . 28
4.2 AntidoteDB node components . . . 29

5.1 Cluster network topology . . . 36
5.2 Throughput–latency plot of the register CRDT and SCRDT . . . 36
5.3 Throughput–latency plot of the set CRDT and SCRDT . . . 38
5.4 Throughput–latency plot of the counter CRDT and SCRDT . . . 39
5.5 Throughput–latency plot of the bounded counter CRDT and SCRDT . . . 40
5.6 Performance comparison of the regular and secure versions of AntidoteDB 43
5.7 Throughput and latency of AntidoteDB throughout the FMKe benchmark with 32 clients . . . 43

6.1 AQL server node . . . 46
6.2 AQL's architecture with the query rewriter module . . . 47
6.3 Extented AQL `CREATE` syntax . . . 47
6.4 Example case where the `COUNTER_INT` data type is mapped to a bounded counter SCRDT . . . 50

7.1 Database schema used for AQL benchmarks . . . 52
7.2 Performance comparison of the different AQL versions using workload NO-COUNTERS . . . 53
7.3 Performance comparison of the different AQL versions using workload COUNTERS . . . 54
7.4 Throughput of AQL throughout the FMKe benchmark . . . 55

List of Tables

2.1 Overview of the explored cryptographic solutions 9

3.1 Cryptographic schemes used by black-box SCRDTs 23

5.1 Latency and throughput of the register CRDT 37
5.2 Latency and throughput of the set CRDT 38
5.3 Latency and throughput of the counter CRDT 39
5.4 Latency and throughput of the bounded counter CRDT 40
5.5 Number of FMKe entities stored in the database prior to running the benchmark 42
5.6 FMKe operations and their relative frequency 42

6.1 Mapping of AQL data types to CRDTs and SCRDTs 49

7.1 Specifications of the Microsoft Azure machines 51
7.2 Workloads used in the AQL benchmarks 53

Chapter 1

Introduction

The idea of a utility based computing industry is not new. In 1961, John McCarthy foresaw computing services during his speech at MIT's centennial celebration [14]:

> *"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility. We can envisage computing service companies whose subscribers are connected to them by telephone lines. Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system. The system could develop commercially in fairly interesting ways. Certain subscribers might offer service to other subscribers".*
>
> — John McCarthy

The vision of utility based computing services has nowadays fully materialized under what we call *cloud computing*. Users are presented with a convenient on-demand access to data storage and computing resources without the need to heavily invest time in maintaining the hardware infrastructure.

A strong motivator for the adoption of the cloud paradigm is the delivery model employed by cloud service providers. Everything is available as a service, *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), *Software as a Service* (SaaS). These models allow an instant or near instant provision of an elastic computing infrastructure and environment. Users can easily scale up and down computing resources according to the current demand. Another benefit of cloud computing is the ability to deploy your application in multiple geographic locations, allowing better reliability and availability guarantees. End users will also experience lower latency and a better overall experience.

1.1 Motivation

Although widely adopted by the industry, the cloud computing paradigm still faces some challenges. Management of the hardware is the responsibility of the cloud providers. Users no longer have full control over their data storage and computing resources like in the traditional computing model, resulting in increased security and privacy risks. The multi-tenant architecture employed by cloud providers can be a problem if the existing isolation mechanisms fail to keep data from one tenant inaccessible to the others. Not only that, but cloud users may not trust the cloud provider's employees to keep their data confidential.

A simple solution is using end-to-end encryption schemes to encrypt data before placing it in untrusted servers. To perform an operation, e.g., search, we need to download all the data, decrypt it, and only then perform the search. Once completed, we encrypt the data again and upload it to the server. This solution is not feasible, it incurs in a huge communication cost for the server and a huge computational cost for the clients, defeating the purpose of using the cloud in the first place. Giving the untrusted servers only the encrypted form of the data is a good approach, the fundamental problem is enabling the servers to perform computations while only having access to the ciphertext. Several encryption primitives are able to address the problem of performing computations on ciphertext. Homomorphic encryption [31], property-preserving encryption [2], searchable encryption [17, 25], and garbled RAM [32] are examples of such primitives. In recent years, a few database systems capable of addressing the problem have emerged, e.g., CryptDB [47], Cipherbase [9] and Arx [46]. They use various techniques in order to not compromise data confidentiality, ranging from the encryption primitives mentioned above to secure hardware.

Widespread access to the internet has led to a large increase in the number of users, many online applications are faced with an ever-increasing demand. Applications must be built with availability and scalability in mind in order to provide a high quality service to their users. These systems are often replicated through multiple geographic locations to achieve that goal.

The previously mentioned encrypted database systems are all centralized or somewhat centralized, adopt strong consistency models and can be very difficult to scale to multiple locations without a significant loss in performance. We want to adopt a weak consistency model where synchronization operations are performed by the server, this rules out solutions that synchronize state with the help of the clients.

1.2 Solution

We wish to develop a scalable and highly available privacy-preserving key-value store. A system capable of scaling seamlessly that does not compromise data confidentiality when running in untrusted machines. To this end, we begin to explore secure conflict-free

replicated data types (SCRDT) and integrate them in the AntidoteDB key-value store. AntidoteDB is a highly available geo-replicated key-value store that supports highly available transactions (HAT), providing strong consistency within a data center and good performance in geo-replicated deployments.

To integrate SCRDTs in the AntidoteDB data store we design and implement an SCRDT client library. This library performs encryption and decryption operations in a transparent way to its user, keeping the CRDT state secure, and only accessible to the client. Despite not having access to the plaintext version of the CRDT state, the server is still able to perform the usual CRDT operations.

Later, we focus our work on AQL, an SQL interface for the AntidoteDB key-value store. We design and implement an AQL version with configurable privacy at the column level, enabling secure queries over the encrypted data and maintaining the necessary data structures to query processing.

Our design allows the application developer to specify how each table column should be encrypted, or even if it should be encrypted at all. This solution provides the users with the choice to encrypt only sensitive data, and thus, not having to pay a high performance penalty.

Finally, we evaluate the performance and scalability of our proposals through experimental evaluation, attaining a comprehensive understanding of the advantages and disadvantages of such systems.

1.3 Contributions

A summary of the main contributions of this thesis is presented below.

- Design and implementation of a privacy-preserving client library that integrates SCRDTs in the AntidoteDB key-value store. This includes the design and implementation of a solution that extends both client libraries and AntidoteDB itself. The SCRDTs make use of homomorphic and property-preserving encryption allowing the synchronization mechanism to work without modifications.
- Enable secure and rich queries with configurable privacy by extending AQL.
- An experimental evaluation comparing the secured and regular versions of AntidoteDB and AQL. We perform not only synthetic benchmarks, but also realistic benchmarks using the FMKe benchmarking tool. FMKe is a standardized benchmark for key-value stores that simulates a realistic workload based on a subsystem of the Danish national health system.

1.4 Outline

The remainder of this document is organized into the following chapters:

- **Chapter** 2 introduces fundamental concepts and work related to the subject of this thesis. The chapter starts with an overview of encryption schemes that allow computations to be performed on ciphertexts. We then explore three proposals of encrypted database systems meant to run on a cloud deemed untrustworthy. Lastly, we present an overview of conflict-free replicated data types, the data objects used by AntidoteDB.
- **Chapter** 3 introduces the concept of SCRDTs. Compared to regular CRDTs, SCRDTs leverage encryption schemes described in Chapter 2 to keep their local state encrypted, while still being able to eventually converge to a common state.
- **Chapter** 4 describes our approach regarding the integration of SCRDTs into AntidoteDB.
- **Chapter** 5 describes and discusses an experimental evaluation of our SCRDTs proposals. We perform micro benchmarks measuring latency and throughput of operations over regular and secure CRDTs. We also make use of the FMKe [62] benchmarking tool to measure the performance and scalability of SCRDTs in a realistic setting.
- **Chapter** 6 is concerned with securing AQL, AntidoteDB's Query Language. Using multiple encryption schemes and our proposed SCRDTs, we build an AQL client that allows AntidoteDB's users to perform rich queries with varying levels of security.
- **Chapter** 7 describes and discusses an experimental evaluation of AQL, where we compare the regular version to our proposed solution.
- The thesis concludes with **Chapter** 8, which summarizes the main findings, contributions, and offers some suggestions for future research.

Chapter 2

Related Work

This chapter introduces fundamental concepts and an overview of existing work relevant to the subject of this thesis. We start with background material on forms of encryption that allow computations on ciphertexts. We explore existing solutions for data stores that do not compromise data confidentiality. Lastly, we provide an overview of conflict-free replicated data types (CRDT).

2.1 Computations on ciphertexts

In this section we will explore solutions to trustworthy cloud computing, ways to ensure the integrity and confidentiality of computations performed in untrustworthy cloud servers.

2.1.1 Homomorphic encryption

First proposed by Rivest et al. in 1978 [52], homomorphic encryption provides the ability to execute computations on encrypted data. The result of such a computation remains encrypted and can be later revealed by the owner of the secret key. There is no limitation in whether a homomorphic encryption scheme is designed to be symmetric (same key used to encrypt and decrypt) or asymmetric (different keys used to encrypt and decrypt) [53].

Acar et al. [1] define an encryption scheme as homomorphic over an operation $\diamond$ if it verifies the following:

$$E(m_1) \diamond E(m_2) = E(m_1 \diamond m_2), \forall m_1, m_2 \in M. \tag{2.1}$$

Where E is an encryption scheme and M denotes the set of all possible messages. Regarding the number of operations and times they may be applied to the ciphertext, homomorphic encryption schemes can be categorized into three types [1].

Partially Homomorphic Encryption

Encryption schemes that allow only one type of operation, e.g., addition or multiplication, to be performed an arbitrary number of times are known as Partially Homomorphic Encryption (PHE) schemes.

PHE schemes were the first to be introduced, starting with RSA [51], which is homomorphic over multiplication [52]. One of the most well-known PHE schemes was introduced by Paillier [45]. The Paillier scheme is homomorphic over addition, however, it offers some additional homomorphic properties over multiplication, more specifically, a ciphertext raised to a constant will decrypt to the product of the plaintext and the constant.

Somewhat Homomorphic Encryption

Still a partial solution to the problem proposed by Rivest et al. in 1978, Somewhat Homomorphic Encryption (SWHE) allows some types of operations, e.g., addition and multiplication, to be performed a finite number of times. The downside is that the size of the ciphertext grows with each operation, rendering SWHE schemes impractical in real-life use. Each time a homomorphic operation is applied, noise is added to the ciphertext, once the noise level reaches a certain threshold, decryption is no longer possible.

Examples of SWHE schemes are those proposed by Sander et al. [55] (allows an unbounded number of AND operations and one OR/NOT operation) and Boneh et al. [18] (unbounded number of additions and one multiplication).

Fully Homomorphic Encryption

The first solution to a Fully Homomorphic Encryption (FHE) scheme was proposed by Gentry [31] in 2009, more than 30 years after Rivest et al. proposed homomorphic encryption. FHE allows an arbitrary number of operations to be performed an arbitrary number of times.

Gentry's solution uses an SWHE scheme alongside a bootstrapping procedure. The bootstrapping procedure consists in applying a decryption function homomorphically to the ciphertext, reducing the noise introduced by the SWHE scheme. A FHE scheme is obtained by recursively applying the bootstrapping procedure.

From a performance point of view, FHE is still prohibitively expensive, however, since Gentry's work, several attempts to further improve FHE have been made [20, 21, 59].

2.1.2 Property-preserving encryption

The ability to perform arbitrary operations on ciphertexts might not be enough, applications may require certain properties of the plaintexts that homomorphic encryption does not provide. Encryption that preserves, or intentionally leaks, some desired property of the plaintexts is known as property-preserving encryption. We will look at two variants, deterministic encryption and order-preserving encryption.

Deterministic encryption

Given a plaintext, a deterministic encryption scheme is one that will always produce the same ciphertext, and thus, provides a way to perform equality checks on ciphertexts. The deterministic property has security implications, it allows an attacker to perform statistical analysis and associate meaning to ciphertexts. If the encryption scheme is asymmetric, the fact that the encryption key is public opens the door to dictionary attacks. Security and privacy definitions in the context of deterministic encryption are presented by Bellare et al. [13].

Order-preserving encryption

Encryption schemes that leak order relations of plaintexts are known as order-preserving encryption (OPE) schemes. Most of the interest in order-preserving encryption comes from the database community [2] because it allows efficient range queries on encrypted data. Similarly to deterministic encryption schemes, users of OPE schemes must be aware of the security implications. Information leaked by OPE should not be overlooked, Boldyreva et al. [16] showed how an OPE scheme leaked the most significant bits of plaintexts.

Motivated by the limited security offered by existing OPE schemes, there has been a recent effort to construct property-preserving schemes with stronger security guarantees. One outcome of such effort is order-revealing encryption (ORE) [19, 22, 40], a generalization of OPE with stronger security guarantees. The work presented by Lewi et al. [40] is very promising. The authors propose a new ORE scheme that is not only more secure than existing OPE schemes, but 65 times faster when encrypting 32-bit integers.

2.1.3 Searchable encryption

Searchable encryption, as its name would suggest, is a solution to the problem of searching on encrypted data. In order to efficiently search over encrypted data, all searchable encryption schemes depend to some degree on deterministic cryptography [30]. Searchable encryption schemes can be designed using either a symmetric or asymmetric setting. In a symmetric cryptography setting, a user encrypts the data before sending it to a remote server, and only he can perform a search operation. Symmetric searchable encryption (SSE) is useful when the user who generates the data is also the one who searches over it. In an asymmetric cryptographic setting, multiple users can encrypt data with the public key, but only the owner of the private key may perform searches over the encrypted data. Useful, for example, in a scenario to provide secure email. Senders send the email encrypted using asymmetric searchable encryption, and then, an email server wants to separate emails that contain a certain keyword from the rest of the emails. The receiver, however, does not wish to provide the email server with the ability to decrypt its emails. Asymmetric searchable encryption enables the email server to perform a search for a keyword without learning any more information about the email.

Symmetric searchable encryption

The first SSE scheme was proposed by Dawn Xiaoding Song et al. [25]. The presented cryptographic scheme is provably secure and guarantees the following properties:

- **Provable secrecy**: The untrusted server cannot learn any information about the plaintext given only the ciphertext.
- **Controlled searching**: Only the user can perform a search operation, the untrusted server cannot search over the ciphertext without the user's authorization.
- **Hidden queries**: Searching for a given keyword does not reveal the keyword to the untrusted server.
- **Query isolation**: The untrusted server cannot learn any information about the plaintext other than the search result.

Given a document with length n, the proposed scheme provides $O(n)$ searching and no extra space and communication overhead. To improve the performance of the linear search operation, Dawn Xiaoding Song et al. propose the use of an encrypted index. This idea is further explored by Goh in 2003 [33], achieving a search complexity of $O(1)$ per document through the use of bloom filters.

Asymmetric searchable encryption

Boneh et al. [17] proposed the first searchable encryption scheme based on asymmetric or public-key cryptography, also known as *Public-key Encryption with Keyword Search* (PEKS). The scheme is based on the concept of a trapdoor. The owner of the private key generates a trapdoor and gives it to an untrusted server; thus enabling it to perform a search operation without being able to learn anything about the plaintext. Compared to symmetric searchable encryption, PEKS has relatively poor performance because of the use of public-key cryptography. Recent work on asymmetric searchable encryption has shown that it is possible to derive a scheme without limiting searches to a single user [27, 48].

2.1.4 Discussion

A cryptographic scheme that provides the ability to perform an arbitrary number of operations on ciphertexts can, arguably, be referred to as the holy grail of encryption. Homomorphic encryption offers this very desirable property, however, its lackluster performance makes it difficult to be considered a viable solution in many scenarios.

It might be the case that the flexibility offered by homomorphic encryption is not necessary. Property-preserving encryption is usable in many scenarios, databases being one as we will see in the next section. Although they are faster than homomorphic encryption,

property-preserving schemes are vulnerable to statistical analysis by nature. The security implications must be carefully taken into account when considering one of these schemes.

Lastly, we saw encryption schemes that offer the ability to search for specific information in ciphertexts. Searchable encryption schemes come in two flavors, symmetric and asymmetric, both provide good security guarantees.

Regarding performance and security guarantees, Table 2.1 provides an overview of the cryptographic solutions explored in this section.

Table 2.1: Overview of the explored cryptographic solutions. Schemes that make it impossible to extract non-negligible information about the plaintext from the ciphertext are considered to have **Good** security.

	Homomorphic	**Deterministic**	**OPE**	**SSE**	**PEKS**
Performance	Slow	Fast	Fast	Fast	Slow
Security	Good	Leaks information	Leaks information	Good	Good

2.2 Secure data stores

In the following section, we briefly present three database systems that address the problem of secure storage and computing in untrusted servers. All of these systems make use of the encryption schemes we just saw to achieve their goal.

2.2.1 CryptDB

CryptDB [47] is a database system that provides confidentiality to applications backed by SQL databases. The system works by placing a proxy server between the application server and the DBMS server. This proxy is responsible for intercepting all SQL queries a client issues and modifying them in a way that allows the DBMS to work on encrypted data. The semantics of the query is preserved and the DBMS never sees plaintext data.

CryptDB addresses two kinds of threats: (*i*) a passive attacker, like a curious database administrator that wants to access confidential data; (*ii*) an attacker that gains control over the infrastructure, like the application server, CryptDB proxy server or database management system (DBMS) server.

Figure 2.1 shows CryptDB's architecture. Rectangular boxes represent processes and rounded boxes represent data. Components with a gray background are added by CryptDB. The dashed arrows represent the attack surface of the threat models. *Threat 1* assumes a passive attacker with access to the DBMS server, snooping on private data. *Threat 2* assumes an attacker that gains complete control over the infrastructure of the application server, CryptDB proxy server and DBMS server.

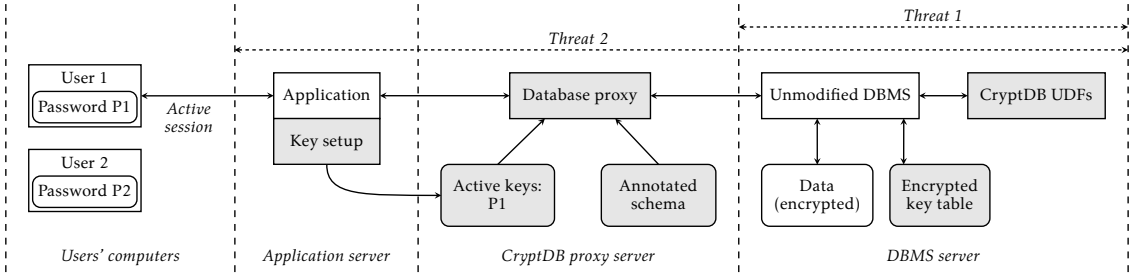

Figure 2.1: CryptDB's architecture—adapted from Popa et al. [47].

An SQL query is processed by the entire infrastructure in four steps, described by Popa et al. as follows:

1. The application issues a query, which the proxy intercepts and rewrites: it anonymizes each table and column name, and, using a master key, encrypts each constant in the query with an encryption scheme best suited for the desired operation.

2. The proxy checks if the DBMS server should be given keys to adjust encryption layers before executing the query, and if so, issues an `UPDATE` query at the DBMS server that invokes a CryptDB specific user-defined function (UDF) to adjust the encryption layer of the appropriate columns.

3. The proxy forwards the encrypted query to the DBMS server, which executes it using standard SQL (occasionally invoking UDFs for aggregation or keyword search).

4. The DBMS server returns the encrypted query result, which the proxy decrypts and returns to the application.

Depending on the type of query issued by the client, the DBMS may need to perform operations that require certain properties from the encryption scheme used to encrypt data. If for example the application requests an equality check on a certain column, the DBMS needs to know which encrypted values correspond to the same plaintext value. This requirement may not be needed on other types of queries, and, in such cases, it is desirable to use a stronger encryption scheme that does not leak which values repeat in a column. CryptDB's approach to handle this challenge is called *adjustable query-based encryption* [47], in which data is encapsulated in multiple layers of encryption. Different layers provide different properties and the system is now able to avoid leaking information when it does not need to. This adjustable encryption mechanism is what allows CryptDB to prevent attackers from accessing private information (threat 1 in Figure 2.1), while still being able to efficiently execute queries. The encryption types used by CryptDB are now briefly described.

Random (RND) RND provides the maximum level of security between those used by CryptDB. It uses a probabilistic encryption scheme, so that when encrypting the same plaintext multiple times, the resulting ciphertext is different.

Deterministic (DET) and order-preserving encryption (OPE) These two types of encryption are described in Section 2.1.2. They use property-preserving encryption schemes that allow CryptDB to perform equality checks, range queries, sorting and aggregations.

Homomorphic encryption (HOM) As described in Section 2.1.1, homomorphic encryption allows one to perform computations on encrypted data. CryptDB uses homomorphic encryption, particularly the scheme proposed by Paillier [45], whenever it needs to perform an addition, e.g., `SUM` or incrementing values.

Join (JOIN and OPE-JOIN) The encryption types JOIN and OPE-JOIN are similar to DET and OPE, respectively. In order to prevent an adversary from learning information about values in different columns, CryptDB uses different keys for different columns when encrypting with the DET and OPE schemes. JOIN and OPE-JOIN enable the server to perform join operations between columns, with the downside of leaking more information than DET and OPE.

Word search (SEARCH) SEARCH allows CryptDB to perform searches on encrypted data, and as a result, perform queries with the `LIKE` operator. It uses the symmetric searchable encryption scheme proposed by Dawn Xiaoding Song et al. [25] and briefly described in Section 2.1.3. The SEARCH encryption type leaks the number of keywords searched.

Even though plaintext data is never exposed under the first threat model, the information leaked by the different types of encryption schemes used by CryptDB should not be overlooked. Akin et al. [3] show that frequency analysis attacks on CryptDB reveal useful information even with a small number of samples.

The second threat model (see Figure 2.1) provides a larger attack surface to an attacker compared to the first threat model. The challenge is to ensure the confidentiality of data in the face of a compromised infrastructure. To mitigate these kinds of threats, CryptDB chains encryption keys to user passwords.

CryptDB follows the application access control policy by requiring developers to annotate their database schema with principals (e.g., users, groups or messages) and the data that each principal has access to. This information is necessary because CryptDB encrypts different data with different keys in order to enforce the access control policy.

Each time a user logs in, it provides its application-level password to the CryptDB proxy server. The proxy server uses the user password to derive the keys of the different encryption layers. The system is now able to process queries on data that the user has access to. Once the user logs out, its key is deleted from the proxy server.

With the approach described above, CryptDB guarantees that an attacker cannot access the data of users that are not logged in while the infrastructure is compromised. Unfortunately, CryptDB cannot guarantee the confidentiality of active users' data.

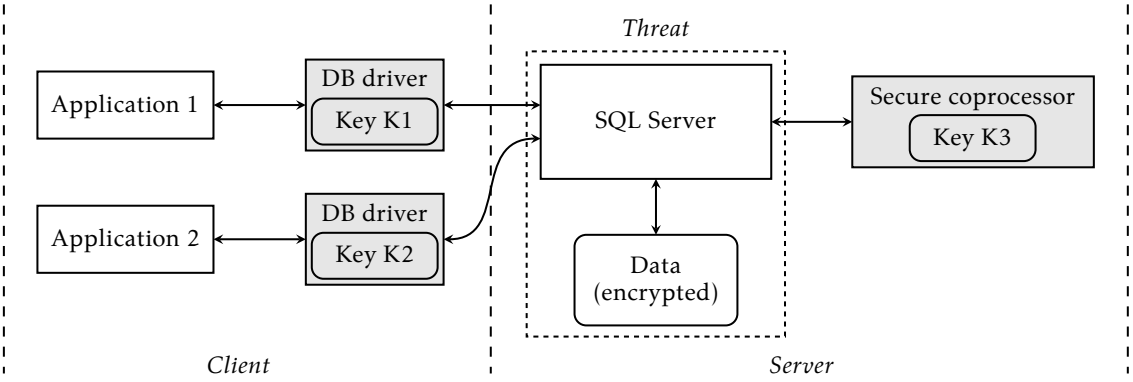

Figure 2.2: Cipherbase's architecture—adapted from Arasu et al. [9]. Rectangular boxes represent processes and rounded boxes represent data. Components with a gray background are added by Cipherbase. The dashed box represents the attack surface of the threat model.

2.2.2 Cipherbase

Cipherbase [9] is a full-fledged SQL database system that provides high performance and data confidentiality through the use of both secure hardware and commodity servers. To avoid performance degradation when there is no need, Cipherbase provides its users with the ability to specify the type of encryption for their data. This feature is called *orthogonal security* because it allows organizations to develop their applications and set their data security goals relatively independently of any performance, scalability, or cost considerations [9]. The supported levels (column granularity) of security options are *no encryption*, *property-preserving* and *homomorphic encryption*, and *strong encryption*.

The Cipherbase system is implemented as an extension to the main components of Microsoft's SQL Server, components from both the client-side and server-side. The threat model considers clients' machines as trustworthy and server machines as untrustworthy, with the exception of a secure coprocessor integrated in the server hardware. Figure 2.2 shows Cipherbase's architecture.

Client-side, Cipherbase extends the database driver used to issue SQL queries and updates in two ways. First, the driver keeps in its local state a secret key used to encrypt query constants[1] before sending them to the server, and to decrypt query results when they arrive from the server. Secondly, the driver, and not the database server, performs query optimization. The client is assumed to be trusted, having him perform query optimization avoids leaking information about the underlying data.

Server-side, a secure coprocessor is integrated into the conventional hardware that runs the database system. The coprocessor consists of one or multiple field-programmable gate arrays (FPGA) that implement a stack machine to process encrypted data. The coprocessor is the only trusted component of the server infrastructure. Whenever the server needs to process a query in which the underlying data is strongly encrypted, the coprocessor is used. The secure coprocessor is able to decrypt and process data and give the results in

[1] The encryption scheme used to encrypt query constants is the same as the security option of the corresponding column.

encrypted form to the server, all without leaking information about the plaintext to the untrusted server. In order to encrypt and decrypt data, a copy of the client's secret key is required. The coprocessor has its own secret key burnt into hardware, which it uses to encrypt and securely store client keys on the disks of the untrusted server. With the help of the coprocessor, the system is able to *simulate* fully homomorphic encryption [9]; efficiently apply a function to a ciphertext as if it was using a FHE scheme.

Due to memory bandwidth constraints, using the secure coprocessor to process data instead of commodity servers, degrades performance. Whenever possible, Cipherbase makes use of property-preserving encryption and homomorphic encryption schemes in order to perform as much computations as possible in untrusted servers. The data security policy defined by the user is what allows Cipherbase to use weaker encryption schemes, as opposed to stronger encryption schemes together with the coprocessor.

Cipherbase focuses primarily on threats to data confidentiality. Similarly to CryptDB, the threat model considers a curious or eavesdropping cloud administrator, that has complete control over the server software and hardware. The exception is the FPGA-based coprocessor, which is considered as trustworthy and vulnerable only to physical attacks to the silicon itself. These types of physical attacks are not considered in the threat model.

2.2.3 Arx

Arx [46] is a recent effort to provide a database system that does not leak information about data, a problem present in older systems like CryptDB [47] and Cipherbase [9]. Proposed by Poddar et al. [46], Arx encrypts data with semantically secure encryption. Defined by Goldwasser et al., being semantically secure means that whatever an eavesdropper can compute about the plaintext given the ciphertext, he can also compute without the ciphertext.

The Arx system introduces two extra components to a database system: a client proxy and a server proxy. These proxies are placed between the application and the database server, both unchanged despite the introduction of the Arx system. The application communicates with the client proxy, which exports exactly the same API as the database server. Meanwhile, the server proxy communicates with the database server by invoking its API, working exactly the same as a regular client. The client proxy rewrites queries received from the application and forwards them to the server proxy if they contain encrypted data, directly to the database server otherwise. In order to encrypt query data and generate cryptographic tokens for the server proxy, the client proxy keeps in its state a master key. Figure 2.3 shows Arx's architecture.

2.2.3.1 Threat model and security

The threat model considers an attacker with complete access to the server-side, consisting of Arx's server proxy and database server, but without access to the client-side. An attacker can extract sensitive information from the system in two ways: (*i*) looking at

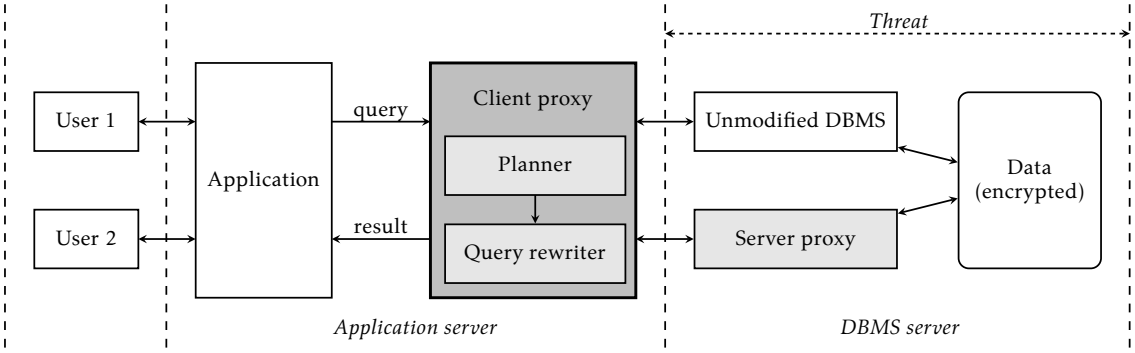

Figure 2.3: Arx's architecture—adapted from Poddar et al. [46]. Boxes with a gray background illustrate components introduced by Arx.

the encrypted data present in the database; (*ii*) observing access patterns during query execution (memory state, *which* and *how many* rows are returned, etc.). Depending on the way sensitive information is obtained, an attacker can be categorized into two types.

Offline attacker An attacker that obtains a complete copy of the encrypted database and analyzes it offline. Many encrypted databases rely on property-preserving encryption schemes to provide efficient equality and order operations, however, these schemes leak information about the underlying data. Recent attacks [28, 36] have shown that it is possible to extract significant information just from the equality and order relations. By using semantically secure encryption, Arx is not vulnerable to offline attacks, it reveals nothing about the data other than size and layout, e.g., number of rows and columns.

Online attacker Besides having access to the encrypted data, an online attacker is able to observe how queries are executed, and although query parameters and constants are encrypted, Arx does not hide metadata or access patterns during execution. In order to mitigate online attacks, Arx limits the amount of information exposed during query execution to only the data involved in the query. The more queries an attacker observes, the more he will learn about the underlying data, for this reason, it is important to detect these types of attacks as soon as possible. In the worst-case scenario, Arx is as bad as an encrypted database that uses property-preserving encryption on the subject of online attacks [35, 37, 38].

The building blocks for Arx's security properties are three semantically secure encryption schemes, `BASE`, `EQ` and `AGG`, and two new database indices, `ArxRange` and `ArxEq`.

`BASE` is a standard probabilistic encryption scheme. `EQ` is a searchable encryption scheme that enables equality checks. `AGG` uses the partially homomorphic encryption proposed by Paillier [45] to perform additions. There is a special case of the `EQ` scheme called `EQunique`, a standard deterministic scheme that is only used when the values of the

underlying data are unique. Applying a deterministic scheme on unique values does *not* leak information.

`ArxRange` enables range and order-by-limit queries by building a tree with a garbled circuit at each node. The garbled circuit performs the comparison between the query and the node's value, revealing neither. If $f(x)$ represents the result of applying the boolean circuit f to input x, a garbled circuit can be used to garble f into $\tilde{f}$ and x into $\tilde{x}$, in such a way that $f(x)$ is revealed but nothing else [32].

Instead of storing the actual value, each node contains the encrypted primary key of the column that contains the value, this way the index does *not* reveal the order relations of the values in the database. To avoid leaking the order in which the values were inserted, the index is built using a history-independent treap [44, 56] instead of a regular search tree.

`ArxEq` is built with searchable encryption and enables equality queries. In the case of unique fields, `ArxEq` encrypts data with `EQunique` and works like a regular database index. In the case of non-unique fields, `ArxEq` works with the help of the client proxy. The client proxy stores a map with an entry for each distinct field value stored in the database. Each entry stores a counter, indicating the number of times the value appears in the database.

Upon insertion of a new row, the system increments the field's counter and encrypts the field's value appended with the counter.

When a client issues a query, the client proxy generates a search token with all the possible encrypted values. If the counter for a value v is given by $map[v]$, the client proxy will generate a search token with the encrypted values for every counter from 1 to $map[v]$. The system can then search through the index with the help of the search token.

Using a counter allows the `ArxEq` index to provide forward privacy, preventing old search tokens from being used to search newly inserted values.

2.2.4 Privacy-preserving NoSQL databases

Research on privacy-preserving databases has been for the most part, centered around SQL databases, the three systems that we just explored attest to that. There are, however, some proposals for privacy-preserving systems around NoSQL databases.

One such work is that of Yuan et al., where the authors propose an encrypted, distributed, and searchable key-value store built on top of the Redis database [63].

Another interesting work is that of Macedo et al., where the authors propose a generic framework that can be used on top of existing NoSQL engines to achieve a privacy-preserving system. The framework exposes a generic NoSQL API with the following operations: *put*, *get*, *delete*, *scan*, and *filter*. The main idea of the framework is to extend NoSQL databases with security mechanisms called CryptoWorkers, which abstract the cryptographic operations of the system. Their proposal offers a modular and flexible design, allowing the underlying system to support multiple techniques with varying performance and security guarantees [43].

2.2.5 Discussion

In this section we saw three solutions to secure databases, systems that provide data confidentiality in the face of security threats.

CryptDB achieves its goal by dynamically adjusting the encryption level of the data. Before insertion, values are protected with multiple layers of encryption. Each layer offers a certain desirable property that enables the system to perform certain types of queries. Of the three systems, CryptDB offers the worst security guarantees. The use of deterministic and OPE encryption schemes make the system vulnerable to statistical analysis attacks that are able to reveal a significant amount of information.

Cipherbase takes an alternative approach, making use of secure hardware to process queries when the underlying data is encrypted. Of the explored systems, Cipherbase is the only one to offer the choice of multiple security options with column granularity. The use of secure hardware allows Cipherbase to offer strong security guarantees, being the only system that provides a solution to hiding access patterns. An unpleasant downside of using exotic hardware solutions is the impossibility of running the system in the commodity machines offered by cloud providers.

The section ends with an overview of the Arx system. A secure database that uses semantically secure encryption to encrypt data. The system meets its goal by introducing two new database indices that enable equality and range queries. Security wise, Arx reveals metadata and access patterns during query execution.

Regarding performance, Arx's authors state that their system is slightly slower than CryptDB when processing equality and range queries. However, Arx is faster when processing aggregations. In the end, both systems have a similar overhead when compared to traditional databases, about 10% [46]. We cannot make a comparison to Cipherbase since the authors do not provide a performance evaluation. However, they affirm that Cipherbase offers almost the same performance as traditional database systems [9].

2.3 Conflict-free replicated data types

It is often the case that distributed systems rely on replication to achieve high availability, low latency and high throughput. Large scale distributed systems that replicate data at different geographic locations, such as distributed databases, usually make a trade-off between read consistency and availability, latency and throughput.

A strong consistency model offering linearizability makes application code simpler and easier to reason about. Strong consistency requires, however, that a quorum of replicas is available, adding a price of higher latency and reduced availability during failures, to the operations of the system. An alternative is to employ a weak consistency model [54], offering higher availability and lower latency with the downside of making the application behavior harder to grasp and code harder to write.

Conflict-free replicated data types proposed by Shapiro et al. [58] provide a simple, theoretically sound approach to eventual consistency, a weak consistency model. CRDTs are abstract data types designed to be replicated. Replicas can be modified without synchronization and are guaranteed to converge to a correct common state. They guarantee *conflict freedom* by leveraging simple mathematical properties, such as monotonicity in a semilattice and commutativity.

The guarantee that all replicas of CRDTs converge to a common state is true if and only if all update operations reach all replicas. To ensure this property, replicas must synchronize. There are two main approaches to this synchronization procedure, state-based synchronization and operation-based synchronization.

2.3.1 State-based synchronization

Replicas of state-based CRDTs synchronize by occasionally sending their local state, which can be modified by update operations, to some other replica. Upon receiving another replica's state, a *merge* function is responsible for merging the local state with the received state.

To prove that state-based CRDTs converge it is sufficient that replicas communicate in a fully connected graph and the CRDT is a monotonic semilattice [58]:

- The set of possible CRDT states form a semilattice.
- Merging the local state with a remote state computes the least upper bound of the two states.
- An update operation produces a new state that is greater than or equal to the original state.

Algorithm 1 shows a state-based implementation of a counter CRDT [57]. A counter supports two update operations, *increment* and *decrement*, and a query operation, *value*. The local state of the CRDT consists of two vectors of integers, each with an entry for each replica. Vector P registers *increment* operations, vector N registers *decrement* operations.

Executing an *increment*, respectively *decrement*, operation in some replica, adds a given value to the entry of vector P, respectively N, assigned to that replica. The value of the CRDT corresponds to the difference of the sum of all entries between vector P and N. Lastly, the *merge* function takes the maximum of each entry.

The correct behavior of this implementation depends on two assumptions: (*i*) entries of vectors do not overflow; (*ii*) the set of replicas is known.

CRDTs with state-based synchronization are easier to reason about, since all the necessary information is captured by the state [57]. However, sending the entire state through the network can be expensive when dealing with large payloads. This weakness can be mitigated with delta-based synchronization [5, 6], later explained in Section 2.3.3.

Algorithm 1 State-based Counter CRDT—adapted from Shapiro et al. [57].

```
1:  payload integer[n] P, integer[n] N          ▷ One entry per replica, n total replicas
2:      initial [0,0, ...,0], [0,0, ...,0]
3:
4:  query value(): integer
5:      ∑_{i=0}^{n-1} P[i] − ∑_{i=0}^{n-1} N[i]
6:
7:  update increment(v):
8:      let g := myID()                         ▷ myID() generates the local replica id
9:      P[g] := P[g] + v
10:
11: update decrement(v):
12:     let g := myID()
13:     N[g] := N[g] + v
14:
15: merge(X, Y): payload Z
16:     forall i ∈ [0..n − 1] do
17:         Z.P[i] := max(X.P[i], Y.P[i])
18:         Z.N[i] := max(X.N[i], Y.N[i])
```

2.3.2 Operation-based synchronization

In operation-based synchronization replicas propagate updates to all replicas. A reliable broadcast channel guarantees that updates are delivered at all replicas in a causal order. Updates delivered out of causal order are said to be concurrent. To ensure convergence, concurrent updates must commute [57]. If updates may be delivered more than once, they must be idempotent.

Operation-based CRDTs do not have a *merge* function; instead they must define two functions for each update, *prepare-update* and *effect-update*. The *prepare-update* function is side effect free and is executed only by the replica that receives the update operation. Its purpose is to generate a representation that encodes the side effects of the update. The *effect-update* function is executed by all replicas, takes the output of *prepare-update* and applies it to the replica's state.

Algorithm 2 specifies the operation-based version of the CRDT counter seen previously. The local state consists of a single integer, which is the return value of the query operation *value*. Both update operations, *increment* and *decrement*, receive an amount as an argument, which is added or subtracted to the state in the *effect-update* function.

Specifying CRDTs with operation-based synchronization can be more complex than with state-based synchronization, since it requires one to reason about history [57]. However, contrary to state-based CRDTs, payloads can be much simpler and concise. One limitation of operation-based CRDTs is the requirement of a reliable dissemination layer with causal delivery and exactly-once semantics [49].

Algorithm 2 Operation-based Counter CRDT—adapted from Preguiça [49]

```
 1: payload integer val
 2:     initial 0
 3:
 4: query value(): integer
 5:     val
 6:
 7: update increment
 8:     prepare-update(v):
 9:         (increment, [v])
10:     effect-update(v):
11:         val := val + v
12:
13: update decrement
14:     prepare-update(v):
15:         (decrement, [v])
16:     effect-update(v):
17:         val := val − v
```

2.3.3 Delta-based synchronization

Delta-based synchronization [5, 6] an attempt to combine the best of both state-based and operation-based CRDTs. Propagating the entire state when only a small part of the CRDT state is modified by an update operation is inefficient. Sometimes propagating the entire state once is better than propagating multiple update operations that modify the same state.

Delta state conflict-free replicated data types (δ-CRDT) define delta-mutators that return a delta-state: a value in the same join-semilattice which represents the updates induced by the mutator on the current state [6]. Propagating the delta-state over the network has a lower cost than propagating the entire CRDT state. Also, the exactly-once delivery requirement is no longer necessary, since a delta-state *is* just a state and not an operation, like in operation-based CRDTs.

2.3.4 Concurrency semantics

Consider a set CRDT that is replicated across many replicas, with multiple *add* and *remove* operations occurring, and being propagated asynchronously between replicas. There is no clear outcome when concurrently adding and removing the same element, these operations are not commutative. One must define reasonable concurrency semantics to avoid ambiguity.

We start by considering the *happens-before* relation [39]. Let a and b be two events, a *happens-before* b, denoted $a \prec b$, if one of the following three conditions is satisfied:

- If a and b are events in the same process, a occurs before b.

- If event a is the sending of a message, b is the receipt of the same message.
- There exists an event c such that, $a \prec c$ and $c \prec b$.

Regarding CRDT operations, $op_1 \prec op_2$, iff the effects of op_1 had been applied in the replica where op_2 was executed initially [49].

For the set CRDT, in the presence of concurrent *add* and *remove* operations, we can define concurrency semantics that gives priority to the *add* operation. This is called the *add-wins* set. In short, an element e belongs to the set if $add(e)$ occurred, and there is no $remove(e)$ operation such that $add(e) \prec remove(e)$. If we define that the *remove* operation takes precedence over the *add* operation, we obtain the *remove-wins* set.

An alternative approach to *add-wins* and *remove-wins* is the *last-writer-wins* (LWW) semantics, which gives priority to updates based on a total order defined among them. LWW sets use some form of timestamp for each element.

2.4 Summary

The idea of performing computations on ciphertexts in not new in the cryptographic community. This chapter described several encryption schemes that enable a system working with encrypted data to do more than storage. Homomorphic encryption schemes are the most flexible, they allow an arbitrary number of operations to be performed on the ciphertext an arbitrary number of times. However, these schemes are prohibitively expensive performance-wise.

Cryptographic schemes less flexible than homomorphic encryption schemes can still be useful. We saw how some database systems, particularly CryptDB and Arx, leverage property-preserving and searchable encryption schemes to provide data confidentiality without compromising functionality. These systems can be incredibly useful, one can benefit from the advantages of cloud computing services, while still keeping sensitive information secure. Cipherbase takes an alternative approach, emulating homomorphic encryption through the use of secure hardware to avoid the performance penalty. However, the use of exotic hardware solutions introduces portability issues, it is no longer possible to run the system in the commodity hardware made available by cloud service providers.

The chapter ends with an overview of CRDTs, data types meant to be replicated, where replicas can be updated without synchronization and are guaranteed to converge to a common state. We saw the two main approaches to CRDTs: state-based CRDTs, and operation-based CRDTs. Implementations of state-based CRDTs are generally simpler, however, sending the entire CRDT state at every synchronization step can be an expensive operation. Operation-based CRDTs reduce the communication cost by sending only update operations to the replicas. On the other hand, they require that updates be delivered to all replicas with exactly-once semantics.

Chapter 3

Secure Conflict-free Replicated Data Types

Initially proposed by Tavares et al. [60, 61], secure conflict-free replicated data types (SCRDT) are a solution to allow update and synchronization operations to occur, while keeping the CRDT state encrypted.

There are two approaches to the implementation of SCRDTs, black-box constructions and homomorphic constructions [12], which were used for the development of the solutions developed in the context of this thesis and presented in Barbosa et al. [12].

3.1 Black-box constructions

Black-box constructions use unmodified standard CRDT implementations with an encryption layer on top.

In some CRDTs, the encryption scheme used must meet certain requirements imposed by the underlying CRDT implementation. Consider the set CRDT specified in algorithm 3. Internal operations must be able to perform equality comparisons between stored values; therefore, the encryption scheme must enable equality comparisons.

A set SCRDT is achieved with a security overlay that encrypts and decrypts data with a deterministic encryption scheme. This security layer is presented in algorithm 4. The set CRDT presented in algorithm 3 is used as is, treated as a black box, hence the name, black-box construction.

The correct behavior of the set SCRDT is guaranteed if the underlying standard implementation is correct. Security and confidentiality of the stored values are guaranteed by the encryption scheme. Note, however, that a deterministic encryption scheme can leak information, as an eavesdropper may recognize known ciphertexts.

Simpler data types, like the register CRDT, do not impose any kind of requirements

Algorithm 3 Operation-based add-wins set CRDT—adapted from Shapiro et al. [58]

```
1:  payload set S
2:      initial ∅
3:
4:  query value(): set                                  ▷ returns the entire set state
5:      {e | (e, u) ∈ S}
6:
7:  query lookup(e): boolean          ▷ returns true if the set contains e, false otherwise
8:      ∃ u : (e, u) ∈ S
9:
10: update add                                          ▷ adds e to the set
11:     prepare-update(e):
12:         (add, [e, unique()])                        ▷ unique() returns a unique identifier
13:     effect-update(e, uid):
14:         S := S ∪ {(e, uid)}
15:
16: update remove                                       ▷ removes e from the set
17:     prepare-update(e):
18:         pre lookup(e)                               ▷ precondition: e ∈ S
19:         (remove, [e, {u | (e, u) ∈ S}])
20:     effect-update(e, uids):
21:         S := S \ {(e, u) | u ∈ uids}
```

Algorithm 4 Security overlay of a set SCRDT using a deterministic encryption scheme Ω, and a standard set CRDT Π_{set}—adapted from Barbosa et al. [12]

```
1:  payload integer key                                 ▷ encryption and decryption key
2:
3:  query value(): set
4:      let S := Π_set.value()
5:      {Ω.decrypt(ciphertext, key) | ciphertext ∈ S}
6:
7:  query lookup(e): boolean
8:      let ciphertext := Ω.encrypt(e, key)
9:      Π_set.lookup(ciphertext)
10:
11: update add(e)
12:     let ciphertext := Ω.encrypt(e, key)
13:     Π_set.add(ciphertext)
14:
15: update remove(e)
16:     let ciphertext := Ω.encrypt(e, key)
17:     Π_set.remove(ciphertext)
```

on the encryption scheme, this is because internal operations do not perform any computation over the stored value. These types of CRDTs can use stronger encryption, like a probabilistic encryption scheme, where through the use of randomness, the encryption of the same message yields different ciphertexts.

Table 3.1 provides a list of CRDTs that allow a black-box construction of a secure counterpart. Note that the map CRDT listed in table 3.1 is a regular map of literals. A more interesting construction is a map of CRDTs, where each key is associated with a CRDT. Similar to the map of literals, a secure map of CRDTs uses a deterministic encryption scheme to encrypt keys, however, since its values are CRDTs and not literals, there is no need to encrypt them. One can use standard or secure CRDTs as values, and in the latter case, encryption is handled by the embedded CRDT itself.

Table 3.1: Cryptographic schemes used by black-box SCRDTs.

Type	Cryptographic Scheme
Set	Deterministic
Map	Deterministic (key) Probabilistic (value)
Register	Probabilistic
Multi-value register	Probabilistic

3.2 Homomorphic constructions

An homomorphic construction uses homomorphic encryption schemes to create secure versions of regular CRDTs. Contrary to black box, this type of construction cannot use unmodified CRDT implementations, due to the type of computations the CRDT performs on stored data.

Consider the operation-based counter CRDT we saw in section 2.3.2, algorithm 2. The update operation *increment* takes a value v and adds it to the stored CRDT state in the *effect-update* function. Now imagine that both the value v and the CRDT state are encrypted. Adding these two encrypted values together is meaningless, as we will not be able to make sense of the resulting sum.

The idea is to use homomorphic or partially homomorphic encryption that allows the same operations to be performed over encrypted data. In the case of the counter CRDT we use the Paillier cryptosystem [45], which is homomorphic over addition. Given two integers a and b, with ciphertexts c_a and c_b, respectively, the product of c_a and c_b will decrypt to the sum of a and b:

$$D(c_a \cdot c_b \bmod n^2) = a + b \bmod n^2 \tag{3.1}$$

Where the function D performs the decryption operation, and n^2 is a parameter of the public key calculated during the key generation.

Algorithm 5 specifies the secure counterpart of the regular counter CRDT. The *effect-update* function of the increment operation was modified to perform the homomorphic addition of ciphertexts.

Algorithm 5 Homomorphic construction of an operation-based counter CRDT using the Paillier encryption scheme [45]

```
1:  payload integer val, boolean spoiled
2:      initial 0, false          ▷ spoiled indicates whether the counter has been incremented or not
3:
4:  query value(): integer                                  ▷ returns the counter state
5:      pre spoiled
6:      val
7:
8:  update increment                                        ▷ increments the counter by v
9:      prepare-update(v, n²):
10:         (increment, [v, n²])
11:     effect-update(v, n²):
12:         if spoiled then
13:             val := val · v mod n²          ▷ homomorphic addition as specified by Paillier [45]
14:         else
15:             spoiled := true
16:             val := v
17:         end if
```

Two caveats of the counter SCRDT that do not apply to the regular version:

- While the regular counter is usually initialized to 0, the secure counter does not know how to represent the encrypted form of the 0 value. To circumvent this issue, we use the first increment operation to initialize the CRDT state. When the variable *spoiled* is false, the *effect-update* function simply sets the CRDT state to the value of v (line 16 in algorithm 5). The downside is that the CRDT state can only be read after the first increment operation.

- The Paillier cryptosystem does not allow us to have an explicit *decrement* operation. In practice, this limitation is easily overcome by representing signed integers in two's complement and using a negative value as the increment delta.

Our counter SCRDT is completed with the addition of the security overlay specified by algorithm 6. As was the case in the set SCRDT, the sole responsibility of this security layer is to encrypt and decrypt values.

Another CRDT that can be made secure through homomorphic construction is the bounded counter. Similarly to the counter SCRDT, the increment operation of the bounded counter SCRDT is modified to perform the homomorphic addition of ciphertexts, however, the counter bounds must hold.

Algorithm 6 Security overlay of a counter SCRDT using the Paillier encryption scheme Θ, and the counter CRDT $\Pi_{counter}$ specified in algorithm 5

```
1: payload integer public, private, n^2          ▷ encryption and decryption keys
2:
3: query value(): integer
4:     let ciphertext := Π_counter.value()
5:     Θ.decrypt(ciphertext, private)
6:
7: update increment(delta)
8:     let ciphertext := Θ.encrypt(delta, public)
9:     Π_counter.increment(ciphertext, n^2)
```

The regular bounded counter models the difference between its value and its bound as a set of rights to execute increment or decrement operations. As long as the CRDT has enough rights available to perform operations, the invariant will hold. This is no longer possible in the secure version, the SCRDT value is encrypted, and the Paillier scheme does not allow us to perform the necessary computations to calculate the set of rights. However, the counter invariant can still be enforced with the help of the client by leveraging transactions. We will expand more on this topic in the next chapter.

3.3 Summary

This chapter introduced the concept of secure CRDTs, providing the necessary tools to answer the core question of this thesis, how to implement a scalable, highly available, and geo-replicated privacy-preserving key-value store.

Leveraging standard CRDT implementations, black-box construction builds SCRDTs by implementing a security overlay that simply deals with the encryption and decryption of data. Homomorphic constructions require not only the security overlay, but also modifications to the underlying CRDT implementation.

We can expect that SCRDTs achieved through homomorphic construction will suffer, performance-wise, when compared to their regular counterpart. This performance difference should not be visible, or at least not as apparent, in black-box SCRDTs, since they make use of faster and less expensive encryption schemes.

Chapter 4

Integration of SCRDTs into AntidoteDB

Having introduced the concept of SCRDTs in the previous chapter, we will now see how we can use them to implement a privacy-preserving key-value store based on AntidoteDB. We start with a brief overview of AntidoteDB's architecture and finish with a detailed explanation of how we integrated SCRDTs into the data store.

4.1 AntidoteDB

AntidoteDB [4, 7] is a highly available, geo-replicated key-value store. AntidoteDB is built using the Erlang programming language, a functional language in which the runtime system offers built-in support for concurrency, distribution and fault tolerance.

AntidoteDB runs on multiple data centers at the same time. In each data center, data is sharded among physical servers with the help of consistent hashing and a distributed hash table with a ring structure. Transactions issued by clients are processed only by the nodes that hold the data [8]. Later and asynchronously, the system propagates updates to the remaining data centers. This design allows AntidoteDB to serve requests and remain highly available even when some servers in a data center fail or network partitions occur.

The system employs Cure [4], a protocol that allows AntidoteDB to offer Highly Available Transactions (HATs) [10] and causal+ consistency, placing itself in a sweet spot in the consistency and availability trade-off.

Despite offering weaker properties than serializable transactions used by traditional ACID databases, HATs still offer much desired guarantees [10] in a context between transactions:

- **Monotonic reads:** After reading an object, subsequent reads on that object will always return the same or a more recent value.

- **Monotonic writes:** Write operations by a process are visible in the same order they were submitted.

- **Writes-follow-reads:** Ensures that if a process observes the effects of a transaction T_1 and later commits transaction T_2, then T_2 will be visible after T_1.

The causal+ consistency model simply combines causal consistency with eventual consistency, ensuring that: (*i*) all clients observe from the system a state that respects the causal relationships between operations; (*ii*) if no new updates occur, replicas eventually converge to the same state.

Regarding data types, AntidoteDB supports operation-based CRDTs [23], available in multiple data types and conflict resolution strategies: *last-writer-wins* register, *multi-value* register, counter (including bounded counter [11]), *enable-wins* and *disable-wins* flag, and *grow-only*, *add-wins*, and *remove-wins* maps and sets.

AntidoteDB's general architecture is depicted in figure 4.1.

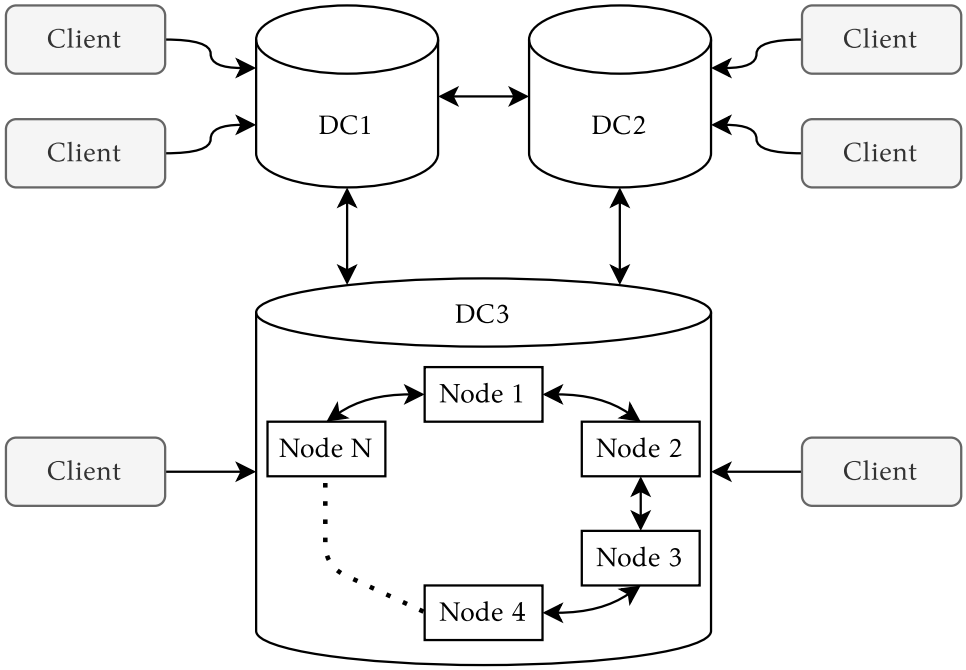

Figure 4.1: Overview of AntidoteDB's general architecture—adapted from AntidoteDB's documentation [8]. The system is organized into multiple clusters or data centers to where clients can connect. Within each cluster, data is sharded among different nodes organized in a ring structure.

Figure 4.2 shows how each node within a data center is organized into the following four components [8]:

- **Transaction manager:** The transaction manager implements the Cure [4] transaction protocol and is responsible for receiving client requests, executing and coordinating transactions, and replying to clients. Interacts with the materializer component to fetch snapshots of the stored data.

- **Materializer:** The materializer is responsible for caching the most recent operations, and snapshots of the objects requested by clients. Upon reception of an operation, the materializer caches it. Whenever a read request is received, the materializer first checks whether pending operations exist in its cache. If so, a new snapshot is created by applying the pending operations to the most recent snapshot. If there are no pending operations in its cache, the materializer simply returns its most recent snapshot of the object. To avoid an infinitely growing number of snapshots, a garbage collection mechanism exists.
- **Log:** The log component maintains the history of object updates. This history is persisted to disk to ensure durability. Used by the materializer on restarts to load the state of objects, or when it is necessary to read an older version of an object than what is available in memory. The log component is also used to resend lost updates to other data centers.
- **InterDC replication:** The InterDC component is responsible for propagating updates from the log to other data centers.

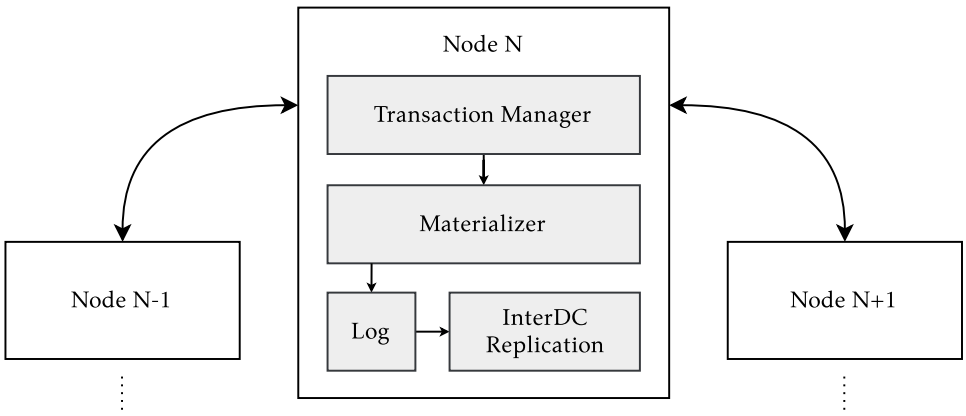

Figure 4.2: AntidoteDB node components—adapted from AntidoteDB's documentation [8].

4.2 Design and implementation

Adding support for SCRDTs to AntidoteDB requires modifications to the AntidoteDB core and to the client libraries. AntidoteDB supports multiple client libraries, implemented in various programming languages. For our prototype we decided to modify both the Erlang [29] and Python [50] client libraries, chosen solely because these would later facilitate our experimental evaluation. In the end, we have a working prototype of AntidoteDB with support for the following SCRDTs:

- Register;
- Multi-value register;

- Set;
- Map of CRDTs;
- Counter;
- Bounded counter.

The register, multi-value register, set, and map SCRDTs are implemented through black-box construction. The counter and bounded counter SCRDTs are implemented through homomorphic construction.

The threat model contemplates two types of attackers: (*i*) a curious database administrator, with access to all the data stored in memory and disk; (*ii*) external attackers capable of eavesdropping communication channels. The database administrator is mitigated through the use of SCRDTs; external attackers are mitigated by securing the communication channels with the Transport Layer Security (TLS) protocol.

4.2.1 Client libraries

As we saw in chapter 3, there are two approaches to implementing SCRDTs, black-box construction, and homomorphic construction. In either approach, an encryption/decryption layer is always necessary. This layer lives in the client library.

A regular AntidoteDB's client library provides a simple interface to work with the different types of CRDTs offered by the data store, e.g., *add* an element to a set, *increment* a counter, etc. To accommodate SCRDTs, we modified the client libraries to encrypt data before sending it to the server, and decrypt data when any arrives from the server. Data is encrypted and decrypted using different encryption schemes, each offering different properties (see table 3.1 in the previous chapter), depending on the type of CRDT used. For the probabilistic scheme, we used AES-OFB with random IVs and 128-bit keys. For the deterministic scheme, we used AES-OFB with fixed IVs[1] and 128-bit keys. And the Paillier cryptosystem with 2048-bit keys for the homomorphic encryption scheme.

Our implementation uses a different encryption key for each data object, this is so that for the same plaintext, different objects will see a different ciphertext, even when using a deterministic encryption scheme. The way our client library generates encryption keys is as follows: (1) a master key is generated; (2) each data object derives its own encryption key using the master key and the key that identifies him in the key-value store.

For the set and counter SCRDTs, the code responsible for encrypting and decrypting data is an implementation of the algorithms 4 and 6, seen in the previous chapter. The remaining SCRDTs implement a similar algorithm. For completeness, algorithms 7, 8, 9, and 10, specify the algorithm implemented by the security code of the register, multi-value register, map, and bounded counter SCRDTs, respectively.

[1] The IV is deterministically derived from the plaintext, e.g., using a cryptographic hash function.

Algorithm 7 Security overlay of a register SCRDT using a probabilistic encryption scheme Φ, and a standard register CRDT $\Pi_{register}$

```
1: payload integer key                                  ▷ encryption and decryption key
2:
3: query value(): T
4:     let ciphertext := Π_register.value()
5:     Φ.decrypt(ciphertext, key)
6:
7: update assign(T v)
8:     let ciphertext := Φ.encrypt(v, key)
9:     Π_register.assign(ciphertext)
```

Algorithm 8 Security overlay of a multi-value register SCRDT using a probabilistic encryption scheme Φ, and a standard multi-value register CRDT $\Pi_{mv-register}$

```
1: payload integer key                                  ▷ encryption and decryption key
2:
3: query value(): [T]
4:     let S := Π_mv−register.value()
5:     [Φ.decrypt(ciphertext, key) | ciphertext ∈ S]
6:
7: update assign(T v)
8:     let ciphertext := Φ.encrypt(v, key)
9:     Π_mv−register.assign(ciphertext)
```

Algorithm 9 Security overlay of a map SCRDT using a deterministic encryption scheme Ω, and a standard map CRDT Π_{map}

```
1: payload integer key                                  ▷ encryption and decryption key
2:
3: query value(): T ↦ CRDT
4:     let S := Π_map.value()
5:     {(Ω.decrypt(ciphertext, key), Π) | (ciphertext, Π) ∈ S}
6:
7: update put(T k, update)
8:     let key_ciphertext := Ω.encrypt(k, key)
9:     Π_map.put(key_ciphertext, update)
10:
11: update remove(T k)
12:     let ciphertext := Ω.encrypt(k, key)
13:     Π_map.remove(ciphertext)
```

Algorithm 10 Security overlay of a bounded counter SCRDT using the Paillier encryption scheme Θ, and an additively homomorphic bounded counter CRDT $\Pi_{b-counter}$

```
1:  payload integer public, private, n^2          ▷ encryption and decryption keys
2:
3:  query value(): integer
4:      let ciphertext := Π_counter.value()
5:      Θ.decrypt(ciphertext, private)
6:
7:  update increment(delta)
8:      let tx := start_transaction()
9:      let v := value()
10:     if v + delta > 0 then
11:         let ciphertext := Θ.encrypt(delta, public)
12:         Π_b−counter.increment(ciphertext, n^2)
13:         tx.commit()
14:     else
15:         tx.abort()
16:     end if
```

Similarly to the counter algorithm presented in 6, the secure bounded counter does not provide an explicit decrement operation, to do so, we simply need to pass a negative delta when incrementing.

The bounded counter security overlay leverages AntidoteDB's transactions to make sure that the limit invariant is kept. Using the Paillier cryptosystem means that the server is no longer able to check whether the increment operation will violate the invariant; thus, this check must now be made by the client. By using the transactional mode offered by AntidoteDB we guarantee that two different transactions cannot concurrently modify the same bounded counter in the local datacenter. If the client checks, inside a transaction, that there are enough rights available to perform an operation, we can guarantee that the bounded counter invariant will hold in the local datacenter.

Note that we never mentioned anything about the conflict resolution policy of the register, multi-value register, set, and map SCRDTs. This was done on purpose, because any conflict resolution policy is valid. From the client's point of view, it is only necessary to encrypt data before sending it to the server, and decrypt it after receiving it from the server. The conflict resolution policy is provided by the underlying CRDT, and the security overlay that exists in the client library is agnostic to that.

4.2.2 Modifications to AntidoteDB

The implementation of the counter and bounded counter SCRDTs is more involved than the remaining SCRDTs. As we saw in the previous chapter, homomorphic construction requires that we implement new CRDTs that make use of a homomorphic encryption scheme.

We extended the AntidoteDB core with an implementation of an operation-based counter and bounded counter CRDTs that use the Paillier cryptosystem, as specified in algorithm 5. These CRDTs were implemented in the Erlang programming language and live in the `antidote_crdt` module [23], following the API used by AntidoteDB's CRDTs [24].

Messages exchanged between the clients and the server are serialized and deserialized using Protocol Buffers[2]. With the addition of new CRDTs, we were required to add new Protocol Buffers' message types to the serialization layer of AntidoteDB.

4.2.2.1 Type alias

Every read and update operation that an AntidoteDB server receives from a client must specify a name and CRDT type. This pair, name and CRDT type, is what tells AntidoteDB the object and CRDT implementation that should be used to perform the operation. Note that the data types and operations are type checked by AntidoteDB, so it is not possible to apply, for example, an *increment* operation to an add-wins set CRDT.

The behavior described above raises a challenge with regards to black-box SCRDTs, because they make use of an existing implementation. Client side, we want CRDTs and SCRDTs to be different types, as an add-wins set SCRDT is different from a *regular* add-wins set CRDT. Server side, the type parameter should always be that of an add-wins set CRDT, even when the client is using the *secure* version.

The above challenge was overcome by extending AntidoteDB with a type alias mechanism. Client libraries provide types for all the different CRDTs and SCRDTs, and send these types to the server. The server, prior to any operation that performs type checking or uses the data type in any way, maps the SCRDT type to the regular CRDT counterpart. The idea is that server side, SCRDT types are simply alias to regular CRDT types.

This typing challenge does not apply to the homomorphic constructed counter and bounded counter. Server side, these two CRDTs have an actual implementation instead of leveraging an existing one.

4.3 Summary

This chapter started by describing AntidoteDB's overall architecture, designed in a way to be scalable and highly available. Later we describe the work that went into adding SCRDTs to AntidoteDB. This work allows us to achieve one of the goals of this thesis, a working prototype of a scalable and highly available privacy-preserving key-value store.

The AntidoteDB modifications described throughout this chapter were submitted and accepted upstream, and are currently open-sourced on GitHub in the various repositories of the AntidoteDB organization[3].

[2] https://developers.google.com/protocol-buffers
[3] https://github.com/AntidoteDB

Chapter 5

SCRDTs experimental evaluation

This chapter presents an experimental evaluation of SCRDTs in AntidoteDB. We perform multiple performance benchmarks to better understand the impact of providing data confidentiality in the AntidoteDB key-value store. To this end, in all experiments, we compare the execution when using regular CRDTs and SCRDTs.

The chapter is split in two parts: (*i*) a set of synthetic benchmarks assessing the latency, throughput, and scalability of the different SCRDT constructions; (*ii*) a realistic benchmark using the FMKe [62] framework.

Our experiments were performed in a cluster of seven nodes, where two were used as servers, and the remaining five were used to run multiple clients in parallel. The server nodes are equipped with an AMD EPYC 7281 CPU, and 128 GiB of RAM each. Both servers are connected to the same network switch, *S1*, with two 10 Gbit/s connections. Three of the five client nodes are equipped with the same hardware as the server nodes. The remaining two client nodes are equipped with two Intel Xeon E5-2620 v2 CPUs each, and 64 GiB of RAM each. These last two nodes are both connected to the same switch, *S2*, with two 1 Gbit/s connections. Switch *S2* is connected to switch *S1* with two 10 Gbit/s connections. The network topology is depicted in figure 5.1.

Saturating the two servers proved to be a difficult task with the number of client machines we had available. For this reason, the AntidoteDB instances running in the server machines were limited to 8 vCPUs, everything else remained unchanged.

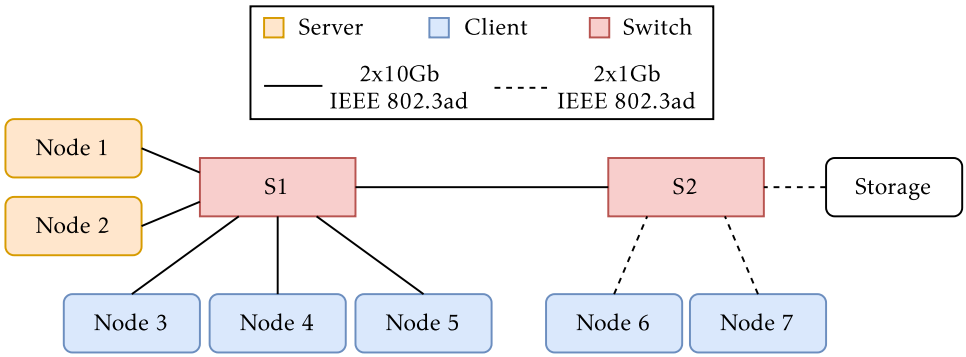

Figure 5.1: Cluster network topology—adapted from DI-CLUSTER [26].

5.1 Synthetic benchmarks

Our synthetic benchmarks consist in running many clients in parallel, each executing operations in a single data type. Each operation is executed in a single data object, randomly chosen among a group of 25 objects. The number of clients is increased until we are able to saturate the servers. The benchmarks are performed by a Python program that uses our modified version of AntidoteDB's Python client library.

We benchmarked the regular and secure variants of the register, set, counter, and bounded counter CRDTs. These four data types cover all the different encryption schemes used: probabilistic, deterministic, and homomorphic.

5.1.1 Register

The register benchmark uses a combination of 50% reads and 50% writes of 2500 random bytes. Table 5.1 shows the mean and 90th percentile latency of each operation, as well as, the overall throughput for the plaintext and secure registers, respectively.

Figure 5.2 shows a throughput–latency plot of the same results. It is clear that the server is saturated in both the plaintext (at around 2500 operations/s) and secure (at around 2200 operations/s) CRDTs.

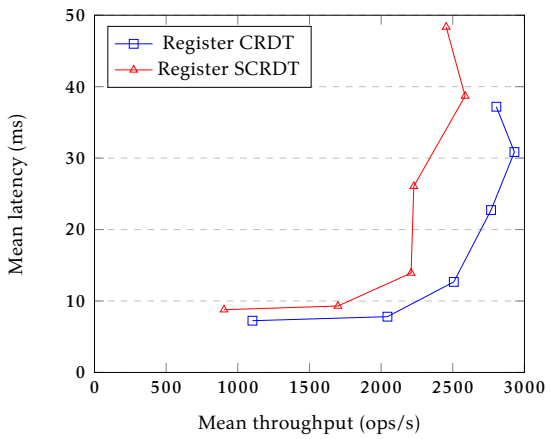

Figure 5.2: Throughput–latency plot of the register CRDT and SCRDT.

Table 5.1: Operation latency in milliseconds (mean and 90th percentile) and overall throughput of the register CRDT and SCRDT.

Clients		Read		Write		Ops/s
		Mean	P90	Mean	P90	
8	Regular	6.48	44.40	8.00	44.96	1102
	Secure	7.81	45.15	9.75	46.04	904
16	Regular	6.77	44.47	8.84	44.80	2044
	Secure	7.42	44.92	11.17	45.73	1699
32	Regular	8.64	44.32	16.70	45.35	2507
	Secure	8.70	45.08	19.14	46.67	2209
64	Regular	11.43	45.07	34.02	50.63	2767
	Secure	14.68	46.46	37.44	51.77	2228
96	Regular	13.78	45.29	47.91	87.93	2930
	Secure	19.42	48.73	57.95	54.17	2586
128	Regular	15.52	46.03	58.86	62.87	2804
	Secure	23.00	50.80	73.69	93.63	2454

The secure register exhibits slightly higher latency and lower throughput. The higher latency values are to be expected, as clients require extra time to encrypt and decrypt data. The lower throughput can be explained by the cryptographic expansion of the data, thus increasing the payload size that the server has to process.

5.1.2 Set

The set benchmark consists of 50% reads, 35% adds and 15% removes. The read operation is over the entire set, meaning that a list containing all the set elements is returned. Table 5.2 and figure 5.3 show the results of the set benchmark. Add operations insert into the set a random 500 byte value.

Once again, results are similar. The secure CRDT exhibits an overall higher operational latency and slightly lower throughput. The server gets saturated at around 1000 operations/s for the plaintext version and 900 operations/s for the secure version.

With a high number of clients, the read operation latency of the secure set becomes increasingly larger than the plaintext counterpart. This can be explained by the ever-increasing size of the set, since 35% of the operations are adds versus only 15% removes. When the benchmark runs with a high number of clients, the read operation takes extra time because clients need to decrypt a large amount of data.

Table 5.2: Operation latency in milliseconds (mean and 90th percentile) and overall throughput of the set CRDT and SCRDT.

Clients		Read		Add		Remove		Ops/s
		Mean	P90	Mean	P90	Mean	P90	
8	Regular	10.36	46.28	11.95	46.19	13.66	46.58	694
	Secure	9.57	45.73	12.13	46.22	13.80	46.38	513
16	Regular	12.73	46.43	30.21	48.05	28.71	47.65	751
	Secure	18.71	49.07	38.29	49.12	45.85	49.85	713
32	Regular	19.35	47.58	85.52	53.11	80.98	59.51	813
	Secure	25.08	56.47	40.97	54.00	42.19	54.55	817
64	Regular	20.53	50.28	96.00	59.28	103.82	63.10	1038
	Secure	94.11	139.02	44.12	61.52	43.92	61.92	959
96	Regular	33.63	52.79	93.31	60.27	95.33	59.74	1258
	Secure	226.12	344.80	62.69	83.19	62.47	82.53	1057
128	Regular	166.77	227.58	96.42	124.36	99.91	137.79	1029
	Secure	227.02	222.87	75.22	100.92	75.82	101.28	921

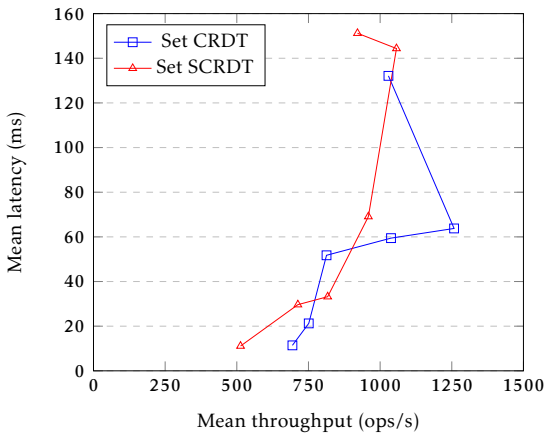

Figure 5.3: Throughput–latency plot of the set CRDT and SCRDT.

5.1.3 Counter

The workload for the counter benchmark consists of 33% reads, 33% increments, and 33% decrements. The results are shown in table 5.3 and figure 5.4.

As expected, the use of homomorphic encryption translates into a massive performance penalty. The Paillier cryptosystem explodes the size of the simple integers used in the plaintext counter into 4096 bit integers in the secure counter, and contrary to the register

Table 5.3: Operation latency in milliseconds (mean and 90th percentile) and overall throughput of the counter CRDT and SCRDT.

Clients		Read		Increment		Decrement		Ops/s
		Mean	P90	Mean	P90	Mean	P90	
8	Regular	6.25	14.34	6.79	44.97	6.88	44.97	1202
	Secure	37.20	72.53	67.98	88.07	66.79	89.38	128
16	Regular	6.61	11.85	7.79	44.96	8.28	45.02	2091
	Secure	62.27	99.60	93.28	121.68	93.45	121.91	193
32	Regular	8.74	44.29	13.17	45.32	14.04	45.50	2643
	Secure	109.17	155.08	152.00	191.36	150.06	189.30	222
64	Regular	12.03	44.92	26.48	47.48	27.61	47.48	2861
	Secure	224.62	301.99	303.13	381.62	301.64	377.47	248
96	Regular	14.68	45.17	38.75	51.47	39.32	50.99	3015
	Secure	407.61	533.01	522.04	638.32	514.78	687.90	207
128	Regular	13.73	45.26	44.46	53.50	43.04	52.20	2875
	Secure	588.28	857.61	728.02	1029.35	704.99	1068.91	204

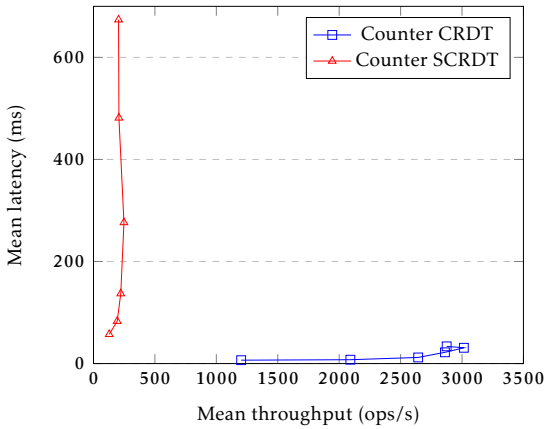

Figure 5.4: Throughput–latency plot of the counter CRDT and SCRDT.

and set SCRDTs, the server now has to perform arithmetic operations with the encrypted data.

Using the secure counter quickly saturates the server. We were only able to obtain a maximum throughput of 248 operations/s, a number more than ten times smaller than the 3015 operations/s of the plaintext counter. Latency values are also significantly higher for the secure counter.

5.1.4 Bounded counter

The benchmark workload of the bounded counter is equal to that of the counter, 33% for each operation. Results are presented in table 5.4 and figure 5.5.

Table 5.4: Operation latency in milliseconds (mean and 90th percentile) and overall throughput of the bounded counter CRDT and SCRDT.

Clients		Read		Increment		Decrement		Ops/s
		Mean	P90	Mean	P90	Mean	P90	
8	Regular	5.82	3.18	6.98	5.43	9.43	46.02	980
	Secure	42.42	78.56	63.86	87.36	61.98	103.66	101
16	Regular	7.50	44.55	14.33	45.41	21.81	46.68	1284
	Secure	69.71	108.99	90.88	122.20	93.80	151.59	177
32	Regular	9.05	44.85	19.35	46.49	27.15	50.92	1545
	Secure	123.46	174.90	149.05	199.96	160.43	242.97	219
64	Regular	13.53	45.47	47.53	103.86	59.42	152.68	1702
	Secure	247.56	360.90	297.46	414.28	320.99	482.06	221
96	Regular	13.96	45.29	66.33	171.72	86.86	251.77	1668
	Secure	359.21	564.96	420.02	630.68	486.59	796.29	219
128	Regular	25.05	48.95	87.41	219.54	108.65	325.05	1637
	Secure	500.20	789.74	567.73	890.62	658.73	1092.26	200

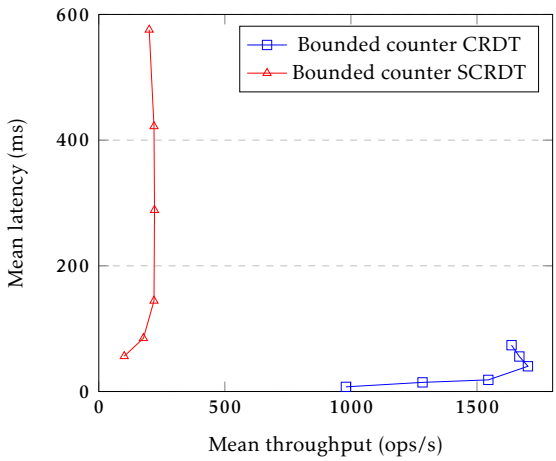

Figure 5.5: Throughput–latency plot of the bounded counter CRDT and SCRDT.

The exhibited behavior is equivalent to the behavior of the counter, as the use of homomorphic encryption leads to a huge performance penalty. The plaintext bounded counter

saturates at around 1600 operations/s, while the secure bounded counter saturates at around 200 operations/s.

When comparing the counter and bounded counter plaintext results, it is clearly visible the cost of having to keep the upper/lower limit invariant of the bounded counter, as the latency increases and the throughput decreases. This behavior is hardly noticeable when comparing the secure counter and secure bounded counter results, explained by the overwhelmingly superior amount of time that the homomorphic operations take relative to keeping the bounded counter invariant.

5.1.5 Discussion

Our results show that the cost of supporting secure CRDT constructions is correlated with the cost of the underlying cryptographic schemes.

SCRDTs like the set and register, which make use of standard symmetric cryptography, result in a relatively low overhead to the system. The price is paid mainly by the client that has to encrypt and decrypt data, whereas the server only has to deal with slightly larger payloads due to ciphertext expansion.

SCRDTs that leverage more novel cryptographic schemes, like the Paillier based counter and bounded counter, result in a very significant overhead to both the client and server. The encryption and decryption operations are orders of magnitude more expensive than in standard cryptographic schemes; moreover, the server is required to perform expensive[1] computations on stored data. The ciphertext expansion is also quite significant, as the simple 32-bit or even 64-bit integers used in the regular CRDTs see their size expanded to 4096-bit. This means larger messages travelling through the network and larger messages for the server to process.

An important aspect of real-world systems is the performance and security trade-off. While security is important, a system designer should be mindful of its cost. We believe that the register and set SCRDTs, and by extension the multi-value register and map SCRDTs, are a feasible solution to systems that want to improve their privacy guarantees. The counter and bounded counter SCRDTs are ok in systems that use them sparingly, as their performance penalty is significant.

5.2 Realistic benchmark

To provide a more realistic performance and scalability evaluation of our prototype, we ran the FMKe benchmark.

FMKe is a benchmark for distributed key-value stores that emulates a real work application. The workload is based on real-life statistics obtained from the Fælles Medicinkort (FMK) system [62], a subsystem of the Danish national health system. We believe FMKe

[1] When compared to the plaintext CRDT.

provides a good use case for a privacy-preserving key-value store, since medical records are, above all, sensitive information.

Table 5.5: Number of FMKe entities stored in the database prior to running the benchmark.

Entity	Number
Hospitals	50
Pharmacies	300
Patients	1000000
Medical Staff	10000
Prescriptions	5000

The benchmark framework is written in the Erlang programming language, and support for AntidoteDB already exists. Adapting the framework to work with our prototype required creating a new AntidoteDB driver using our Erlang client library. To store the different entities and model the relations between them, FMKe uses only three types of CRDTs: register, set, and map. Prior to the benchmark the database is populated with data, namely, hospitals, pharmacies, patients, medical staff, and prescriptions. The number of each entity is presented in table 5.5.

The benchmark focuses on prescription management—table 5.6 shows the list of operations and their relative frequency. We ran the benchmark for ten minutes, multiple times, each time with a different number of clients. The hardware setup is the same as the one described at the beginning of this chapter.

Table 5.6: FMKe operations and their relative frequency.

Operations	Frequency
Get pharmacy prescriptions	27%
Get prescription medication	27%
Get staff prescriptions	14%
Create prescription	8%
Get processed prescriptions	7%
Get patient	5%
Update prescription	4%
Update prescription medication	4%
Get prescription	4%

5.2.1 Results

The overall throughput and latency results are shown in figure 5.6.

Both versions reach maximum throughput at 32 clients, from that point onwards, the server is saturated and we observe a decrease in throughput and a huge increase

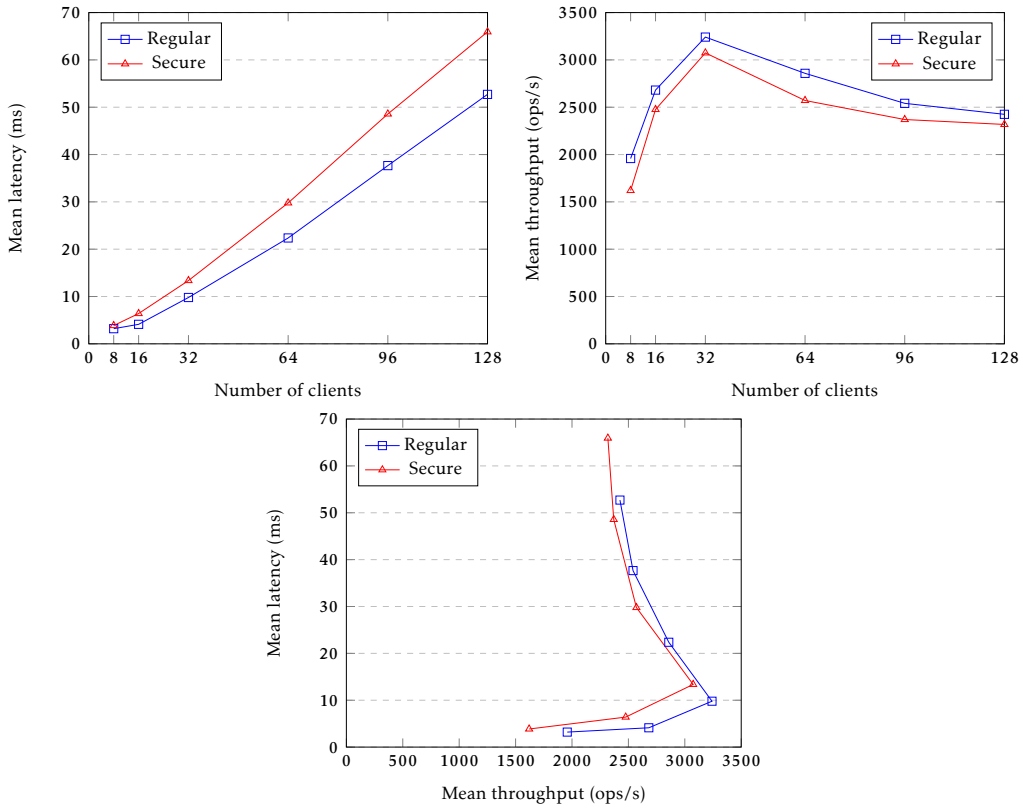

Figure 5.6: Performance comparison of the regular and secure versions of AntidoteDB.

in latency. The top right graph shows that the throughput difference between the two versions remains relatively stable across the board, meaning that SCRDTs do not hinder the scalability of the database. Seeing that the benchmark is only using SCRDTs with standard encryption schemes, the overhead of the secure version is relatively small, and the same behavior was observed in the synthetic benchmarks section.

Figure 5.7 gives us insight into the stability of the system. The data shown is relative to the run with 32 clients, where we periodically measured, once each 10 seconds, the mean throughput and latency. The two graphs show that both the throughput and overall latency of the system remain stable throughout the benchmark.

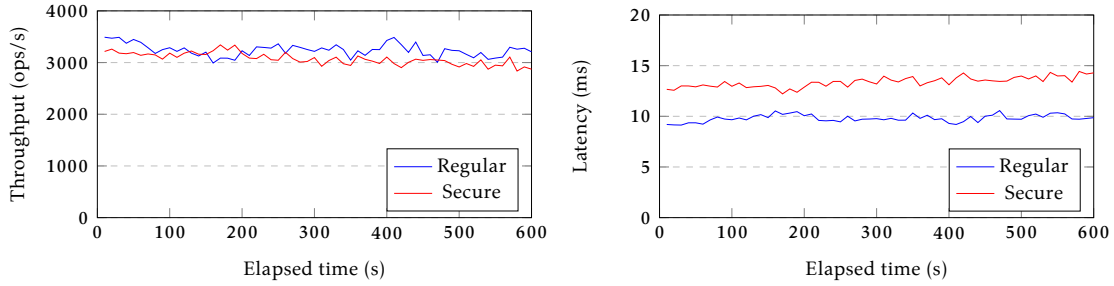

Figure 5.7: Throughput and latency of AntidoteDB throughout the FMKe benchmark with 32 clients.

5.3 Summary

Our experiments show that our design has a variable throughput penalty. SCRDT designs, that make use of standard cryptographic schemes, are a feasible solution to improve data privacy and confidentiality. Homomorphic based SCRDTs, however, incur in a huge performance penalty and are not a feasible solution to systems that make heavy use of them.

For a benchmark that models a real world application, that includes no data types requiring homomorphic based SCRDTs, the overhead is low, which makes SCRDTs a practical solution for providing privacy-preserving features in such application.

Chapter 6

Antidote Query Language

Apart from a privacy-preserving data store, we also wish to enable secure and rich queries by extending Antidote Query Language (AQL), an SQL interface for the AntidoteDB key-value store. This chapter starts by presenting AQL's architecture and inner workings, and finishes with a description of the AQL's modifications we performed in order to add support for secure queries.

6.1 AQL's architecture

Being built as a layer on top of AntidoteDB means that AQL's architecture follows AntidoteDB's architecture. The system runs on multiple data centers at the same time, each with multiple nodes organized in a ring structure [42]. An AQL module is added to each server node that runs an AntidoteDB instance. Clients issue queries directly to the AQL module, which then communicates with AntidoteDB to process those queries. Figure 6.1 shows an overview of AQL's architecture.

AQL offers application developers an SQL interface with varying degrees of consistency. The main idea is to relax SQL consistency when possible, while keeping stricter consistency when necessary [41]. Note that unlike NoSQL databases, AQL, even under relaxed consistency, still enforces primary key, foreign key, and check constraints.

The adopted approach is to let the developer specify in the database schema how and when SQL consistency can be relaxed. This is achieved by extending the regular SQL data definition language with options to configure concurrency semantics, e.g. what should be the outcome of concurrent transactions that perform an update and delete operation over the same table row. By omitting the concurrency semantics specification, AQL assumes a *no concurrency* approach, where strict SQL consistency is enforced through the use of locks at the AntidoteDB level, and requiring coordination among replicas [41].

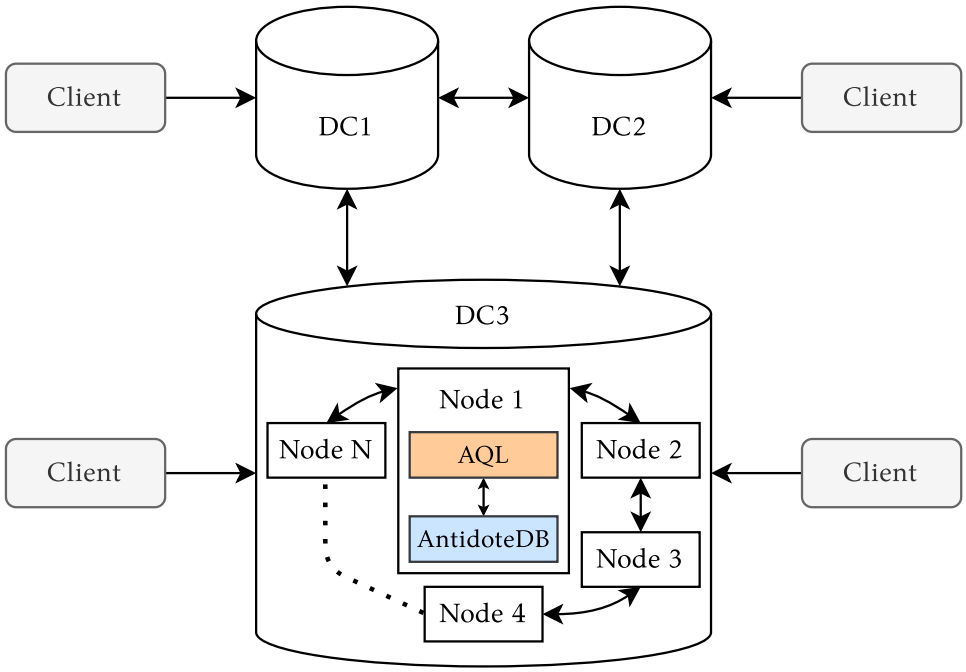

Figure 6.1: Overview of AQL's architecture—adapted from Lopes [42]. Data centers run multiple server nodes in a ring structure. Each node runs the AQL module and an AntidoteDB instance.

6.2 Design and implementation of a secure AQL

Adding support for secure queries to AQL requires modifying the AQL module and the way client libraries work. A regular AQL client simply sends SQL commands to a server and receives the response, showing it to the user. Our implementation consists in the addition of an additional layer to the client library, a query rewriter. The client library is extended to keep encryption keys, and the task of encrypting and decrypting sensitive data is performed by the query rewriter. The flow of a secure SQL command is as follows:

1. The application writes a regular SQL command and uses the interface of the client library to send it to the server.

2. Prior to sending the SQL command to the server, the client library uses the query rewriter module to rewrite the command in a way that all sensitive data is encrypted.

3. The rewritten command is sent to the server and processed.

4. When the reply is received by the client library, the result is decrypted.

5. The decrypted result is returned to the user.

The encryption and decryption of operations is transparent, and final applications should only see plaintext values. Figure 6.2 shows the new AQL architecture, with the addition of the query rewriter module.

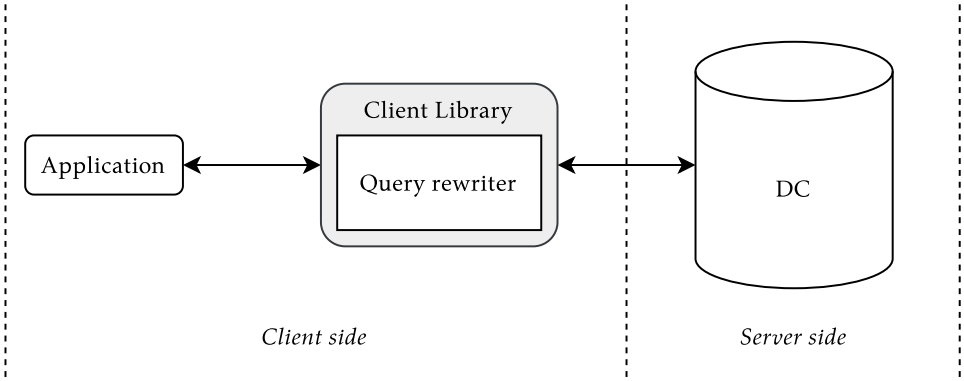

Figure 6.2: AQL's architecture with the query rewriter module.

The fundamental problem is executing queries over encrypted data. AQL's `WHERE` predicate supports multiple comparison operators, =, <>, <, <=, >, and >=. Furthermore, multiple predicates can be combined with the `AND` and `OR` keywords. When the query rewriter encrypts data, it must do so in a way that the server is still capable of performing those operations and reach the correct result.

We handle the above problem by letting the user specify the encryption scheme of table columns at the same time the database schema is specified. For this purpose, we extend AQL's DDL syntax with encryption annotations. The new `CREATE` statement syntax is shown in figure 6.3.

```
create              → CREATE [conflict_resolution] TABLE identifier (
                            identifier column_type [encryption_type] [constraint],
                            ...
                      );

conflict_resolution → UPDATE-WINS | DELETE-WINS

column_type         → INTEGER | BOOLEAN | COUNTER_INT | VARCHAR

encryption_type     → ENC | DTENC | OPENC | HMENC

identifier          → STRING
```

Figure 6.3: Extended AQL `CREATE` syntax, with the addition of four new table column annotations: `ENC`, `DTENC`, `OPENC`, and `HMENC`.

The following encryption annotations were added:

- **`ENC`:** Uses a strong and probabilistic encryption scheme to encrypt the column's data. Good when privacy is the only concern as it only allows to store and retrieve data, the server cannot perform any kind of operation on this data.
- **`DTENC`:** Stands for deterministic encryption, allowing the server to process equality operations on encrypted data, e.g. `WHERE` clause with the = and <> operators.

- **OPENC:** Uses an order-preserving encryption scheme which allows processing comparison operations with the =, <>, <, <=, >, and >= operators.

- **HMENC:** Stands for homomorphic encryption and is usable with columns of the type `COUNTER` or `BCOUNTER`. Uses the Paillier encryption scheme to process increment operations.

This design enables the system to support configurable privacy at the column level, since the application developer is able to choose the best encryption technique for each column. The ommission of the encryption annotation means that the column's data will not be encrypted. It may be the case that encryption is not necessary and there is no need to pay the performance price.

A limitation of this design is that the user must be aware of the types of queries and operations that will be performed on the data. By using a probabilistic encryption scheme on a column, it will be impossible to process queries with a `WHERE` clause that filter on that same column. If the requirements change, all columns need to be encrypted again somehow.

6.2.1 Query rewriter

The query rewriter is responsible for encrypting any data that will be stored, or used in an operation with a column that was marked as encrypted during the schema definition. To encrypt data, the query rewriter must know the correct encryption scheme of each column, which is achieved by requesting metadata to the server. The metadata table stores multiple attributes that describes tables and columns, among this information is the encryption scheme of each column. The metadata table is populated by the AQL server module upon the table creation.

SQL commands must be rewritten in the following cases:

- In `CREATE` statements with a `DEFAULT` clause, the default values must be encrypted if their associated column is encrypted.

- All values in `INSERT` statements must be encrypted accordingly.

- New values in `UPDATE` statements must be encrypted accordingly. Also any value that goes in the optional `WHERE` clause and is used in a comparison with an encrypted column.

- In `SELECT` statements with a `WHERE` clause, values that will be used in a comparison with an encrypted column must be encrypted accordingly. The response from the server will also have to be decrypted when it arrives.

6.2.2 Indexing system

A database index allows, generally, to process queries faster, and is a common tool used by application developers to improve the performance of their system. AQL supports both primary and secondary indexes, implemented as native AntidoteDB operation-based CRDTs. Each table has a corresponding primary index, and secondary indexes are only created if the corresponding create index statement is executed.

Seeing that the indexing system plays a major role in query performance, it is important to make sure that this mechanism is still available to application developers even when the indexed table data is encrypted.

Both the primary and secondary index CRDTs are designed around two data structures, a sorted set and an index tree. Both need to support *lookup* and *range* operations, which means that primary key columns and any other column that is going to be indexed, must be encrypted with `OPENC`, or not be encrypted at all. The index consistency is only ensured if the underlying data allows equality and range comparisons to be performed.

6.2.3 Modifications to the AQL module

Most of the work required to support secure queries is done by the client library, our design requires only one modification to the server side AQL module.

AQL saves each column value in a single AntidoteDB object, a CRDT, whose data type is related to the AQL column type. The fact that some column values may now be encrypted, requires changing the mapping of AQL column type to CRDT data type.

AQL supports four data types: `VARCHAR`, `INTEGER`, `BOOLEAN`, and `COUNTER_INT`; table 6.1 shows how each AQL data type is mapped to CRDTs, when the column data is not encrypted, and SCRDTs when the column data is encrypted.

In all but one case, it is simply a matter of using the secure version of the CRDT, e.g., if the column type is `VARCHAR`, we use a register CRDT when data is not encrypted, and a register SCRDT when data is encrypted. The only exception is the `BOOLEAN` data type, where the plaintext version uses the flag CRDT, and the encrypted version uses a register CRDT since there is no secure counterpart of the flag CRDT.

Table 6.1: Mapping of AQL data types to CRDTs and SCRDTs. The data type `COUNTER_INT` is mapped to the bounded counter CRDT/SCRDT only when the column is declared with check constraints in the database schema.

AQL Data Type	CRDT	SCRDT
`VARCHAR`	Register	Register
`INTEGER`	Register	Register
`BOOLEAN`	Flag	Register
`COUNTER_INT`	Counter Bounded counter	Counter Bounded counter

Note that the COUNTER_INT data type can be mapped to two different CRDTs, counter or bounded counter. By default, the counter CRDT is used, the only instance the COUNTER_INT data type is mapped to the bounded counter CRDT is when the column is declared with check constraints. E.g., AQL will map the *views* column to a bounded counter SCRDT when using the CREATE statement listed in figure 6.4.

```
CREATE UPDATE-WINS TABLE foo (
    id INTEGER PRIMARY KEY,
    views COUNTER_INT HMENC CHECK (views > 0)
);
```

Figure 6.4: Example case where the COUNTER_INT data type is mapped to a bounded counter SCRDT.

6.3 Summary

This chapter started with a light description of AQL's architecture. Implemented as a module on top of AntidoteDB, AQL enables applications to interface with the AntidoteDB key-value store using SQL-like statements.

We end the chapter by describing the design and implementation of a solution that enables securing AQL with configurable privacy at the column level.

The implementation of the work described in this chapter is available on GitHub.[1,2]

[1] AQL secure client library: https://github.com/mrshankly/aqlc
[2] AQL server module: https://github.com/mrshankly/secure-aql

Chapter 7

AQL experimental evaluation

This chapter presents an experimental evaluation of the secure AQL prototype described in the previous chapter. The presented analysis tries to assess the performance and scalability cost of supporting configurable privacy in AQL.

7.1 Setup

The experiments were run on the Microsoft Azure cloud computing service using two types of machines. Two E8-2s v3 machines were used as servers, and four F32s v2 machines were used to ran multiple clients in parallel. The specifications of each machine are presented in table 7.1.

Table 7.1: Specifications of the Microsoft Azure machines.

Machine	vCPUs	RAM	Temporary Storage
E8-2s v3 (servers)	2	64 GiB	128 GiB
F32s v2 (clients)	32	64 GiB	256 GiB

The E8-4s v3 machines belong to the memory optimized family of virtual machines (VM), while the F32s v2 machines belong to the compute optimized family of VMs.

7.2 Benchmark

We perform benchmarks in three different configurations: (NO-ENC) a system where data is not encrypted; (SOME-ENC) a system where some data is encrypted (data belonging to indexed columns is not encrypted); (ALL-ENC) a system where all columns are encrypted, including indexed columns. In the first configuration, where data is not encrypted, the

client library simply relays the query to the server and waits for a response, the query rewriter module is never used. In the case of the second and third configurations, where some or even all data is encrypted, the client library uses the query rewriter module. The database model is shown in figure 7.1.

```
CREATE UPDATE-WINS TABLE one (
    id INTEGER OPENC PRIMARY KEY,
    name VARCHAR ENC,
    age INTEGER ENC,
    address VARCHAR DTENC,
    email VARCHAR DTENC
);

CREATE UPDATE-WINS TABLE two (
    id INTEGER OPENC PRIMARY KEY,
    title VARCHAR ENC,
    views COUNTER_INT HMENC
);
```

Figure 7.1: Database schema used for AQL benchmarks. The columns *name*, *age*, *address*, *email*, *title*, and *views* are encrypted in the configurations SOME-ENC and ALL-ENC. The primary key columns are encrypted only in the ALL-ENC configuration.

All configurations are evaluated in two different workloads, one without operations that involve `COUNTER_INT` columns (we will call this workload NO-COUNTERS), the other with operations that involve `COUNTER_INT` columns (we will call this workload COUNTERS). In total, there are seven possible operations:

- **Select1:** The *select1* operation fetches data from table *one*. Basically a `SELECT` query with a simple `WHERE` clause. This operation uses the encryption types `ENC` and `DTENC`, and `OPENC` when indexed columns are encrypted.
- **Select2:** The *select2* operation fetches data from table *two*. This operation uses the encryption types `ENC`, `DTENC` and `HMENC`, and `OPENC` when indexed columns are encrypted.
- **Update1:** Updates an existing record in table *one*. This operation uses the same encryption types as the *select1* operation.
- **Update2:** Updates an existing record in table *two*. This operation uses the same encryption types as the *select2* operation.
- **Insert1:** Performs an `INSERT` statement, adding a new row to table *one*. Uses the same encryption types as the *select1* operation.
- **Insert2:** Performs an `INSERT` statement, adding a new row to table *two*. Uses the same encryption types as the *select2* operation.

- **Delete:** Performs a `DELETE` statement with a `WHERE` clause consisting in an equality on the primary key column, deleting only a single record from a table. This *delete* operation only uses the `OPENC` encryption type, and only when indexed columns are encrypted.

The relative frequency of each operation is shown in table 7.2.

Table 7.2: Workloads used in the AQL benchmarks, showing the relative frequency of each operation.

Workload	Select1	Select2	Update1	Update2	Insert1	Insert2	Delete
NO-COUNTERS	60%	0%	20%	0%	15%	0%	5%
COUNTERS	55%	5%	15%	5%	13%	2%	5%

Finally, we employ the same strategy as in the SCRDTs experiments, increase the number of clients until we are able to saturate the servers. Each benchmark ran for 2 minutes, and prior to each run, all database tables are populated with 10000 records.

7.3 Results

The results of the benchmark using workload NO-COUNTERS are shown in figure 7.2, in the form of a throughput-latency plot.

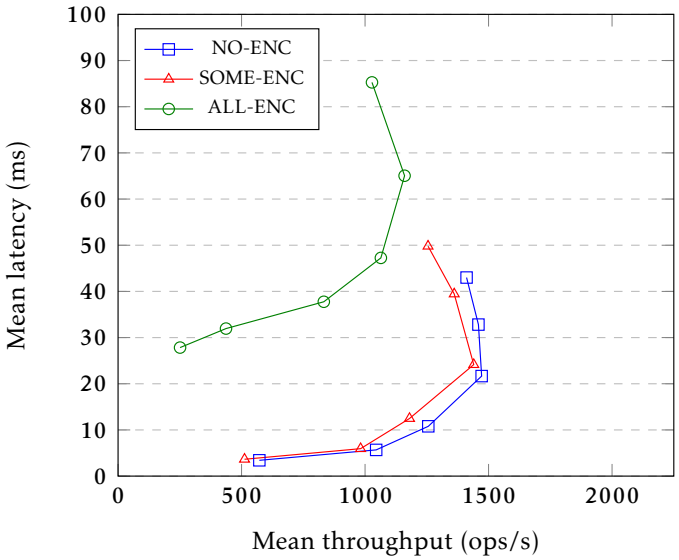

Figure 7.2: Performance comparison of the different AQL versions using workload NO-COUNTERS.

The SOME-ENC configuration is able to follow very closely the NO-ENC configuration. This is expected, as SOME-ENC only uses standard probabilistic and deterministic encryption schemes with relatively low performance overhead. In both configurations the

server is saturated between the 1400 and 1500 operations per second. This shows that the addition of the query rewriter module by itself does not result in a noticeable performance overhead, as the cost can be mapped to the cost of the underlying encryption scheme.

The ALL-ENC configuration uses an order-preserving encryption scheme to encrypt indexed columns, leading to a higher operational latency. The performance price is paid mostly by the client, because even though it takes more clients to saturate the server, the server is saturated at around 1250 operations per second, relatively close to the saturation point of the NO-ENC and SOME-ENC configurations. The slightly lower throughput value of the ALL-ENC configuration can be explained by the ciphertext expansion of data, resulting in larger messages sent to the server and larger messages for the server to process. The order-preserving encryption scheme we use is an implementation of the algorithm proposed by Boldyreva et al. [15], which has an expansion ratio of 2, since ciphertexts are always twice the size of plaintexts.

The results of the benchmark using workload COUNTERS are shown in figure 7.3. As expected, the introduction of `COUNTER_INT` columns, encrypted with an homomorphic encryption scheme, results in a higher performance overhead, and the SOME-ENC configuration is unable to match the results of the NO-ENC configuration. In the NO-ENC configuration the server is saturated at around 2000 operations per second, while in the SOME-ENC configuration the server saturates at around 1600 operations per second. The introduction of a small percentage of operations that make use of homomorphic encryption results in a 20% reduction of the mean throughput. This result agrees with the conclusion we reached in chapter 5, that homomorphic encryption is feasible when its use is occasional, and beyond that, the performance penalty is prohibitive.

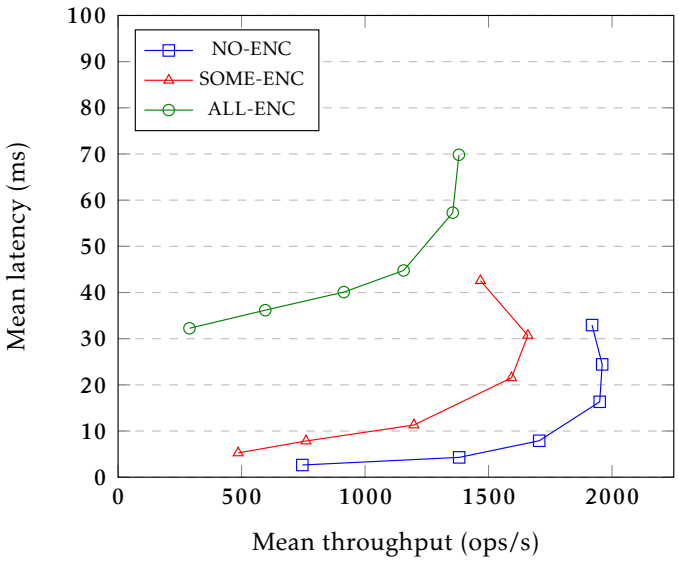

Figure 7.3: Performance comparison of the different AQL versions using workload COUNTERS.

Regarding the ALL-ENC configuration, we can see that the results are similar to those

observed in workload NO-COUNTERS. The overall system experiences an increase in operation latency, with more clients being necessary to saturate the server. The performance penalty is mostly paid by the client since the saturation point of the server is relatively close to the saturation point observer when using the SOME-ENC configuration.

7.3.1 FMKe

Similarly to what we did in the experimental evaluation of SCRDTs, we intended to provide a more realistic performance and scalability analysis of our AQL prototype by running the FMKe benchmark. However, the obtained results do not allow us to provide such analysis.

The FMKe benchmark creates a few tables with a composite primary key, something that AQL does not currently support. This by itself is not a problem, we can create such tables by using a surrogate key, e.g., an integer column with an artificial value that has no meaning attached to it. The problem is that by doing that we lose the index, and we cannot create a secondary index on the appropriate columns, because AQL does not currently support indexes with composite keys.

These tables are used to save relations between medical prescriptions and multiple other entities, such as patients, pharmacies, and medical staff. Since the benchmark revolves around medical prescriptions, these tables are constantly being queried, and without the appropriate index, queries can take more than half a second to be processed. The throughput of the overall system quickly deteriorates after running the benchmark for only a few seconds, eventually reaching values of less than 10 operations per second. Figure 7.4 shows the throughput evolution of the system throughout a 10 minute benchmark.

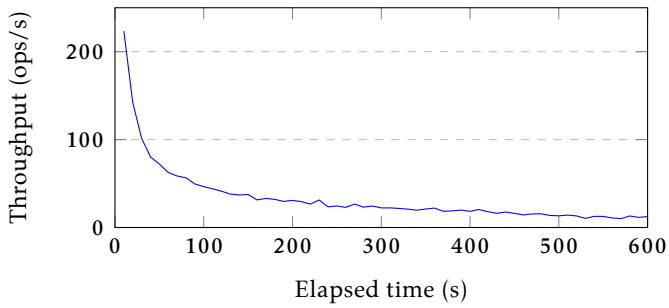

Figure 7.4: Throughput of AQL throughout the FMKe benchmark.

With the server being completely saturated in just a few seconds, we were not able to perform any kind of comparison between a regular and secure version of AQL.

7.4 Summary

This chapter presented an experimental evaluation of our AQL prototype with configurable privacy. We focused on three different configurations: the first one, used as a base line, a

database where data is not encrypted; a second setup, where all but the indexed columns are encrypted; and finally the third one, a database where all data is encrypted. Each configuration ran two types of workloads, one without `COUNTER_INT` columns, and one with `COUNTER_INT` columns. With these two workloads we tried to assess the performance penalty of using an homomorphic encryption scheme.

With these experiments, our intent is to show that an application developer can choose between multiple degrees of privacy-preserving settings, with different advantages and disadvantages. From the obtained results we observed that a system can provide some level of privacy with a very small performance penalty if it only makes use of standard encryption schemes. Regarding the use of homomorphic encryption schemes, although expensive, they should not be ruled out, as they might still be an appropriate solution for systems that wish to increase their privacy and confidentiality at the cost of some performance.

Securing indexed columns results in a slight increase of operational latency to the clients. However, the server only experiences a small performance decrease, as our experiments show that the saturation point remained relatively close to one in a system where indexed columns are left unprotected.

Chapter 8

Conclusion

This thesis proposes to implement a scalable, highly available, and geo-replicated privacy-preserving key-value store. To this end, we show that by leveraging cryptographic schemes and existing CRDT designs, it is possible to achieve secure, privacy-preserving, and semantically correct CRDTs. Our work also demonstrates how these secure CRDTs can be used to improve the security and privacy of an SQL interface for a NoSQL database, where the underlying data objects are represented by CRDTs.

We offer a detailed discussion of our implementation of SCRDTs in the distributed key-value store, AntidoteDB. We detail the modifications required to client libraries and the server itself, as well as the differences and potential caveats compared to the regular, unmodified version. Additionally, we demonstrate how we modified AQL to support rich queries with customizable security and privacy levels.

Our experiments show that SCRDTs that use standard cryptographic schemes result in a small, to moderate performance overhead, but with the upside of significantly improving the privacy and confidentiality of the system. However, constructions that make use of more novel cryptographic schemes significantly deteriorate the system performance.

Despite the fact that not all SCRDT constructions are optimal performance-wise, we show that our prototype is a viable solution in some real-world scenarios. We run our prototype with the FMKe benchmark, simulating the access patterns of a production system used in the health care industry. Results show that our solution is relatively close to the insecure counterpart in terms of performance, and that the scalability of the system is not affected.

Regarding AQL, our experiments show that our solution is capable of accommodating varying levels of privacy with a performance overhead proportional to that of the underlying cryptographic schemes. Even in the case of a database where all the data is encrypted, the server saturation point remains relatively close to that of systems where a

more relaxed privacy setting is used.

8.1 Future work

For future work we suggest looking into AQL's indexing mechanism. As we described in the experimental evaluation chapter, we were unable to run the FMKe benchmark because of the lack of composite key indexes. These types of indexes are an important mechanism that real world applications regularly use to improve their performance.

We believe index performance can be further improved by looking into better encryption schemes, instead of using order-preserving encryption (OPE), future work should consider order-revealing encryption (ORE). Compared to OPE, ORE is more secure, does not expand the ciphertext size by a significant amount, and is generally more efficient.

Another venue to pursue is the use of searchable encryption, which should allow the database to support operators such as `LIKE` in a secure manner.

8.2 Publications

Part of the results originated from this thesis have been accepted for publication:

- **Secure Conflict-free Replicated Data Types.** Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela and Nuno Preguiça. In proceedings of the *22nd International Conference on Distributed Computing and Networking*. ICDCN 2021. January 2021.

Bibliography

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3214303. URL: https://doi.org/10.1145/3214303.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. "Order Preserving Encryption for Numeric Data". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 563–574. ISBN: 1581138598. DOI: 10.1145/1007568.1007632. URL: https://doi.org/10.1145/1007568.1007632.

[3] I. H. Akin and B. Sunar. "On the Difficulty of Securing Web Applications Using CryptDB". In: *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. Dec. 2014, pp. 745–752. DOI: 10.1109/BDCloud.2014.75.

[4] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98.

[5] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-Based CRDTs by Delta-Mutation". In: *Networked Systems*. Ed. by A. Bouajjani and H. Fauconnier. Cham: Springer International Publishing, 2015, pp. 62–76. ISBN: 978-3-319-26850-7.

[6] P. S. Almeida, A. Shoker, and C. Baquero. "Delta state replicated data types". In: *Journal of Parallel and Distributed Computing* 111 (2018), pp. 162–173. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.08.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731517302332.

[7] *AntidoteDB*. Last accessed: 2020-10-21. URL: https://www.antidotedb.eu/.

[8] *AntidoteDB's documentation*. Last accessed: 2020-10-27. URL: https://antidotedb.gitbook.io/documentation/.

[9] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. "Orthogonal Security With Cipherbase". In: *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*. Jan. 2013. URL: https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/.

[10] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Highly available transactions: Virtues and limitations". In: *Proceedings of the VLDB Endowment* 7.3 (2013), pp. 181–192.

[11] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants". In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2015, pp. 31–36. DOI: 10.1109/SRDS.2015.32.

[12] M. Barbosa, B. Ferreira, J. Marques, B. Portela, and N. Preguiça. *Secure Conflict-free Replicated Data Types*. Cryptology ePrint Archive, Report 2020/944. https://eprint.iacr.org/2020/944. 2020.

[13] M. Bellare, A. Boldyreva, and A. O'Neill. "Deterministic and Efficiently Searchable Encryption". In: *Advances in Cryptology - CRYPTO 2007*. Ed. by A. Menezes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 535–552. ISBN: 978-3-540-74143-5.

[14] J. Belzer, A. Holzman, and A. Kent. *Encyclopedia of Computer Science and Technology: Volume 12 - Pattern Recognition: Structural Description Languages to Reliability of Computer Systems*. Taylor & Francis, 1979. ISBN: 9780824722623.

[15] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. "Order-Preserving Symmetric Encryption". In: *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*. Ed. by A. Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 224–241. DOI: 10.1007/978-3-642-01001-9_13. URL: https://doi.org/10.1007/978-3-642-01001-9%5C_13.

[16] A. Boldyreva, N. Chenette, and A. O'Neill. "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by P. Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 578–595. ISBN: 978-3-642-22792-9.

[17] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. "Public Key Encryption with Keyword Search". In: *Advances in Cryptology - EUROCRYPT 2004*. Ed. by C. Cachin and J. L. Camenisch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 506–522. ISBN: 978-3-540-24676-3.

[18] D. Boneh, E.-J. Goh, and K. Nissim. "Evaluating 2-DNF Formulas on Ciphertexts". In: *Theory of Cryptography*. Ed. by J. Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 325–341. ISBN: 978-3-540-30576-7.

[19] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. "Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation". In: *Advances in Cryptology - EUROCRYPT 2015*. Ed. by E. Oswald and M. Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 563–594. ISBN: 978-3-662-46803-6.

[20] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *ACM Trans. Comput. Theory* 6.3 (July 2014). ISSN: 1942-3454. DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600.

[21] Z. Brakerski and V. Vaikuntanathan. "Efficient Fully Homomorphic Encryption from (Standard) LWE". In: *SIAM Journal on Computing* 43.2 (2014), pp. 831–871. DOI: 10.1137/120868669. eprint: https://doi.org/10.1137/120868669. URL: https://doi.org/10.1137/120868669.

[22] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. "Practical Order-Revealing Encryption with Limited Leakage". In: *Fast Software Encryption*. Ed. by T. Peyrin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 474–493. ISBN: 978-3-662-52993-5.

[23] *CRDT implementations for AntidoteDB*. Last accessed: 2020-11-10. URL: https://github.com/AntidoteDB/antidote_crdt.

[24] *CRDT interface used by AntidoteDB*. Last accessed: 2020-11-10. URL: https://github.com/AntidoteDB/antidote_crdt/blob/master/src/antidote_crdt.erl.

[25] Dawn Xiaoding Song, D. Wagner, and A. Perrig. "Practical techniques for searches on encrypted data". In: *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*. May 2000, pp. 44–55. DOI: 10.1109/SECPRI.2000.848445.

[26] *DI-CLUSTER*. Last accessed: 2020-11-15. URL: https://cluster.di.fct.unl.pt/docs/technical/.

[27] C. Dong, G. Russello, and N. Dulay. "Shared and searchable encrypted data for untrusted servers". In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2008, pp. 127–143.

[28] F. B. Durak, T. M. DuBuisson, and D. Cash. "What Else is Revealed by Order-Revealing Encryption?" In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1155–1166. ISBN: 9781450341394. DOI: 10.1145/2976749.2978379. URL: https://doi.org/10.1145/2976749.2978379.

[29] *Erlang client for AntidoteDB*. Last accessed: 2020-10-29. URL: https://github.com/AntidoteDB/antidote-erlang-client.

[30] B. L. d. S. Ferreira. "Privacy-preserving efficient searchable encryption". PhD thesis. 2016. URL: http://hdl.handle.net/10362/19797.

[31] C. Gentry. "A Fully Homomorphic Encryption Scheme". PhD thesis. Stanford, CA, USA, 2009. ISBN: 9781109444506.

[32] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. "Garbled RAM Revisited". In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by P. Q. Nguyen and E. Oswald. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 405–422. ISBN: 978-3-642-55220-5.

[33] E.-J. Goh. *Secure Indexes*. Cryptology ePrint Archive, Report 2003/216. 2003. URL: http://eprint.iacr.org/2003/216/.

[34] S. Goldwasser and S. Micali. "Probabilistic encryption". In: *Journal of Computer and System Sciences* 28.2 (1984), pp. 270–299. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(84)90070-9. URL: http://www.sciencedirect.com/science/article/pii/0022000084900709.

[35] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. "Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1067–1083. DOI: 10.1109/SP.2019.00030.

[36] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. "Leakage-Abuse Attacks against Order-Revealing Encryption". In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 655–672. DOI: 10.1109/SP.2017.44.

[37] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. "Generic Attacks on Secure Outsourced Databases". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1329–1340. ISBN: 9781450341394. DOI: 10.1145/2976749.2978386. URL: https://doi.org/10.1145/2976749.2978386.

[38] M. Lacharité, B. Minaud, and K. G. Paterson. "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage". In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, pp. 297–314. DOI: 10.1109/SP.2018.00002.

[39] L. Lamport. "Time, Clocks and the Ordering of Events in a Distributed System". In: *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.* (July 1978), pp. 558–565. URL: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

[40] K. Lewi and D. J. Wu. "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1167–1178. ISBN: 9781450341394. DOI: 10.1145/2976749.2978376. URL: https://doi.org/10.1145/2976749.2978376.

[41] P. Lopes, J. Sousa, V. Balegas, C. Ferreira, S. Duarte, A. Bieniusa, R. Rodrigues, and N. Preguiça. *Antidote SQL: Relaxed When Possible, Strict When Necessary*. 2019. arXiv: 1902.03576 [cs.DB].

[42] P. M. S. Lopes. *Antidote SQL: SQL for Weakly Consistent Databases*. 2018. URL: http://hdl.handle.net/10362/68859.

[43] R. Macedo, J. Paulo, R. Pontes, B. Portela, T. Oliveira, M. Matos, and R. Oliveira. "A Practical Framework for Privacy-Preserving NoSQL Databases". In: *36rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 2017.

[44] M. Naor and V. Teague. "Anti-Persistence: History Independent Data Structures". In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC '01. Hersonissos, Greece: Association for Computing Machinery, 2001, pp. 492–501. ISBN: 1581133499. DOI: 10.1145/380752.380844. URL: https://doi.org/10.1145/380752.380844.

[45] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by J. Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[46] R. Poddar, T. Boelter, and R. A. Popa. "Arx: An Encrypted Database Using Semantically Secure Encryption". In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1664–1678. ISSN: 2150-8097. DOI: 10.14778/3342263.3342641. URL: https://doi.org/10.14778/3342263.3342641.

[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 85–100. ISBN: 9781450309776. DOI: 10.1145/2043556.2043566. URL: https://doi.org/10.1145/2043556.2043566.

[48] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. "Building Web Applications on Top of Encrypted Data Using Mylar". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 157–172. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/popa.

[49] N. Preguiça. *Conflict-free Replicated Data Types: An Overview*. 2018. arXiv: 1806.10254 [cs.DC].

[50] *Python client for AntidoteDB*. Last accessed: 2020-10-29. URL: https://github.com/AntidoteDB/antidote-python-client.

[51] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[52] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[53] R. Rothblum. "Homomorphic encryption: From private-key to public-key". In: *Theory of cryptography conference*. Springer. 2011, pp. 219–234.

[54] Y. Saito and M. Shapiro. "Optimistic Replication". In: *ACM Comput. Surv.* 37.1 (Mar. 2005), pp. 42–81. ISSN: 0360-0300. DOI: 10.1145/1057977.1057980. URL: https://doi.org/10.1145/1057977.1057980.

[55] T. Sander, A. Young, and M. Yung. "Non-Interactive CryptoComputing For NC1". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS '99. USA: IEEE Computer Society, 1999, p. 554. ISBN: 0769504094.

[56] R. Seidel and C. R. Aragon. "Randomized search trees". In: *Algorithmica* 16.4 (Oct. 1996), pp. 464–497. ISSN: 1432-0541. DOI: 10.1007/BF01940876. URL: https://doi.org/10.1007/BF01940876.

[57] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[58] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 9783642245497.

[59] N. P. Smart and F. Vercauteren. "Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes". In: *Public Key Cryptography – PKC 2010*. Ed. by P. Q. Nguyen and D. Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 420–443. ISBN: 978-3-642-13013-7.

[60] J. Tavares, N. Preguiça, and B. Ferreira. "SCRDTs: Tipos de Dados Seguros e Replicados sem Conflitos". In: *INForum*. Coimbra, Portugal, 2018.

[61] J. d. S. Tavares. *Secure Abstractions for Trusted Cloud Computation*. 2018. URL: http://hdl.handle.net/10362/61548.

[62] G. Tomás, P. Zeller, V. Balegas, D. Akkoorath, A. Bieniusa, J. Leitão, and N. Preguiça. "FMKe: A Real-World Benchmark for Key-Value Data Stores". In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '17. Belgrade, Serbia: Association for Computing Machinery, 2017. ISBN: 9781450349338. DOI: 10.1145/3064889.3064897. URL: https://doi.org/10.1145/3064889.3064897.

[63] X. Yuan, X. Wang, C. Wang, C. Qian, and J. Lin. "Building an Encrypted, Distributed, and Searchable Key-Value Store". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: Association for Computing Machinery, 2016, pp. 547–558. ISBN: 9781450342339. DOI: 10.1145/2897845.2897852. URL: https://doi.org/10.1145/2897845.2897852.